10-27-1994

An Analysis of Approaches to Efficient Hardware Realization of Image Compression Algorithms

Kamran Iravani
Portland State University

THESIS APPROVAL

The abstract and thesis of Kamran Iravani for the Master of Science in Electrical and Computer Engineering were presented October 27, 1994, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS:

[redacted]

Dr. Marek Perkowski, Chair

[redacted]

Dr. Michael Driscoll

[redacted]

Dr. Bradford Crain
Representative of the Office of Graduate Studies

DEPARTMENT APPROVAL:

[redacted]

Dr. Rolf Schaumann, Chair
Department of Electrical Engineering

* *

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by [redacted] on 6 January 1995

ABSTRACT

An abstract of the thesis of Kamran Iravani for the Master of Science in Electrical and Computer Engineering presented October 27, 1994.

Title: An analysis of approaches to efficient hardware realization of image compression algorithms.

In this thesis an attempt has been made to develop a fast algorithm to compress images. The Reed–Muller compression algorithm which was introduced by Reddy & Pai [3] is fast, but the compression factor is too low when compared to the other methods. In this thesis first research has been done to improve this method by generalizing the Reed–Muller transform to the fixed polarity Reed–Muller form.

This thesis shows that the Fixed Polarity Reed–Muller transform does not improve the compression factor enough to warrant its use as an image compression method.

The paper, by Reddy & Pai [3], on Reed–Muller image compression has been criticized, and it was shown that some crucial errors in this paper make it impossible to evaluate the quality and compression factors of their approach.

Finally a simple and fast method for image compression has been introduced. This method has taken advantage of the high correlation between the adjacent pixels of an image. If the matrix of pixel values of an image is divided into bit planes from the Most

Significant Bit (MSB) plane to the Least Significant Bit (LSB) plane, most of the adjacent bits in the MSB planes (MSB, 2nd MSB, 3rd MSB and 4th MSB) are the same. Using this fact a method has been developed by Xoring the adjacent lines of the MSBs planes bit by bit, and Xoring the resulting planes bit by bit. It has been shown that this method gives a much better compression factor, and can be realized by much simpler hardware compared to Reed–Muller image compression method.

AN ANALYSIS OF APPROACHES TO EFFICIENT HARDWARE REALIZATION OF IMAGE COMPRESSION ALGORITHMS

by
Kamran Iravani

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
1994

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I wish to express my appreciation to the individuals who gave assistance to this work. First, I would like to thank my advisor and the chair of my thesis committee, Dr. Marek Perkowski, who provided many suggestions vital to this research as well as guidance and support. I would also like to thank the other members of my committee, Dr. Michael Driscoll and Dr. Bradford Crain.

I want to express my gratitude to staff of the Electrical Engineering department at Portland State University for their help and support all the time.

My appreciation is extended to my friends, especially Andisheh Sarabi, Fardin Ansari and Stoney Vintson, for their help throughout the completion of this thesis.

Finally, I wish to express my appreciation to my family for their support during all stages of my life.

Portland, Oregon Kamran Iravani

October 1994

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

In recent decades there has been a great interest in image data compression. This is due to the fact that digital representation of the images usually requires a large number of bits, while in many applications it is desired to represent the image with a fewer number of bits.

Depending on the applications, different compression methods have been developed. For example in some applications it is important not to lose any information while compressing the images. In other words the reconstructed image is desired to be exactly the same as the original one. In other applications it is desirable to compress the image as much as possible. It is only necessary that the quality of the image be good enough for visual or machine analysis, and loss of some in formation about the image is acceptable.

Based on these applications there are two general categories of image compression techniques, lossy image compression and lossless image compression. In lossy image compression the reconstructed image is not exactly the same as the original but the compression factor is high, while in lossless image compression the reconstructed image is exactly the same as the original one but the compression ratio is not as high. Therefore there is going to be a trade off between the compression factor and the quality of the image.

Another important factor in image compression techniques is the speed of the process. For example in some applications it is important for the process to be fast, while in other

applications it is only required that the image is compressed regardless of how fast it is done. The speed of the process is directly related to the number of operations needed for the compression, and also to the hardware realization of the compressor. Sometimes to obtain a good compression factor the circuit has to do many operations which makes the process slow.

Therefore to choose a compression technique for a specific type of images, four important factors are considered: The compression factor, the quality of the image, the speed of the process, and the cost of the technique.

For example, NASA has used Differential Encoding[11] to monitor the surface of the earth. Differential Encoding is a lossless image compression technique which gives a good compression factor for these type of images. This method is based on calculating the difference between the pixel values of adjacent lines in the image.

Run–Length Coding [12] is another lossless method used to compress images. In this method each sequence of the pixel values is described by two numbers. One number shows the value of pixels and the other number shows the run–length. This method is specially used for the images that contain only two different values. For example Run–Length Coding is implemented for flood maps where the presence of water is represented by white and the absence of water is represented by black..

In the early 1980s a joint ISO/CCITT committee known as JPEG (Joint Photographic Experts Group) [15],[16] began working on a new method of image compression that would greatly outperform the more conventional compression techniques. This method has become the first international compression standard for continuous–tone still images. This standard supports a wide variety of applications. The JPEG specification consists

of several parts including the specification for both lossless and lossy compression. For lossless compression the Predicting/Adaptive coding technique is used. In this method some information about upcoming pixels is predicted based on the previous pixels seen. The most interesting part of the JPEG specification is the technique for lossy compression. This method is based on the Discrete Cosine Transform (DCT), and for continuous–tone images, it gives a high compression factor with good quality. The only disadvantage of this method is that the process is slow because it requires a large number of operations.

To obtain a fast method of compression, research has been done on compression using the Haar Transform [2]. Karpovsky [6] has shown that since expansion coefficients of logical functions in Haar series depend on the local behavior of this function, such an order of arguments can be found that gives the minimum number of non–vanishing coefficients. This property has been used to compress the image. This method is a lossless method but the compression factor is not very high. Although the process requires less number of operations compared to the JPEG algorithm, this method did not become popular because it has a low compression factor.

Another approach to obtain a fast method for image compression was made by Reddy & Pai [3] who used the Reed–Muller transform for this purpose. In this method the pixel matrix of image is divided into eight matrices from the Most Significant Bit (MSB) plane to Least Significant Bit (LSB) plane, and on each plane the Reed–Muller transform is performed, and next the run–length coding [1],[4] is used. Although this method is fast it does not give a good compression factor. The published paper by Reddy & Pai contains several mistakes which make their results and conclusions unacceptable.

It was the purpose of this thesis to improve the Reed–Muller image compression by using the fixed polarity Reed–Muller transform [7]–[10], [11], which is a more general case. It was shown that the result is still poor and a good compression factor cannot be obtained. Solely for the purpose of comparison, a new method was introduced which is very simple and very fast with a much higher compression factor compared to the Reed–Muller transform. This method has taken advantage of the correlation between the adjacent lines. This method is based upon Xoring the lines and planes, and gives a good compression factor.

In this thesis some of the methods, from above, have been investigated, and the main goal has been to develop a fast method with reasonable quality and good compression factor.

In chapter 2, image compression methods in general have been explained and all the steps involved have been overviewed in more detail. These steps are Mapping, Quantizing and Coding for lossy compression, but for lossless compression they are just Mapping and Coding. Because, as it will be explained, the quantization step causes an information loss.

In chapter 3, Run–Length Coding has been explained in detail. This method of coding is used in most of the image compression techniques. Since all of these coding methods use Huffman coding [5], the creation of Huffman codes is explained first. Then one dimensional Run–Length Coding (RLC) [1] and two dimensional Run–Length Coding which consists of Relative Address Coding (RAC) [4] and Relative Element Address Designate (READ) [1] are explained. Two dimensional coding techniques are usually more efficient than one dimensional coding, but they are more complicated.

In chapter 4, the JPEG algorithm for lossy compression has been explained in detail. In this chapter the Discrete Cosine Transform (DCT) has been explained and it has been shown that the compression based on DCT requires a large number of operations, although the compression factor is very good.

In chapter 5, the compression based on Haar transform has been investigated. In this chapter the algorithm to find the best order of arguments to give the minimum number of nonvanishing coefficients has been explained step by step. Then image compression using this technique is presented.

In chapter 6, compression based on Reed–Muller transform is discussed. First positive polarity Reed–Muller and fixed polarity Reed–Muller expansion of a Boolean function is explained. Next, the algorithm for compressing image by this transform is investigated. Finally, comments on the local behavior of Reed–Muller transform and on the possibility of finding a good permutation of input vector which gives the minimum number of nonvanishing coefficients, are made.

In chapter 7, a discussion on the paper published by Reddy & Pai [3] on Reed–Muller image compression is made, and the mistakes in this paper are exposed and analyzed. The mistakes are in both transformation and coding which makes the results of this paper totally unacceptable.

In chapter 8, a new compression method which is realized by Xoring the lines and planes is explained, and the reasons for having a relatively good compression factor are given.

In chapter 9, the experimental results of both image compression methods, the fixed polarity Reed–Muller transform, and the Xoring of lines and planes, are presented. These

results show that the latter method is a better one from the point of view of the quickness, hardware realization, and compression factor. In this method the compression factor in some planes is more than twice as high as that of the former method. The compression has been applied three different natural pictures which are continuous–tone still images.

In chapter 10, the conclusion of this thesis is given.

Chapter 2

An Overview of Image Compression Methods

2.1. Introduction

In this chapter image compression methods are explained in general, and all the steps involved in the process of compressing an image are discussed. This provides the general background needed for the following chapters about image compression techniques.

2.2. Image compression

In general, an image compression technique consists of three successive steps: Mapping, Quantizing and Coding, which can be modelled by a block diagram from Fig. 2.1. In this diagram the matrix of pixel values is shown as a vector form (one dimensional array), while it can be also two–dimensional.

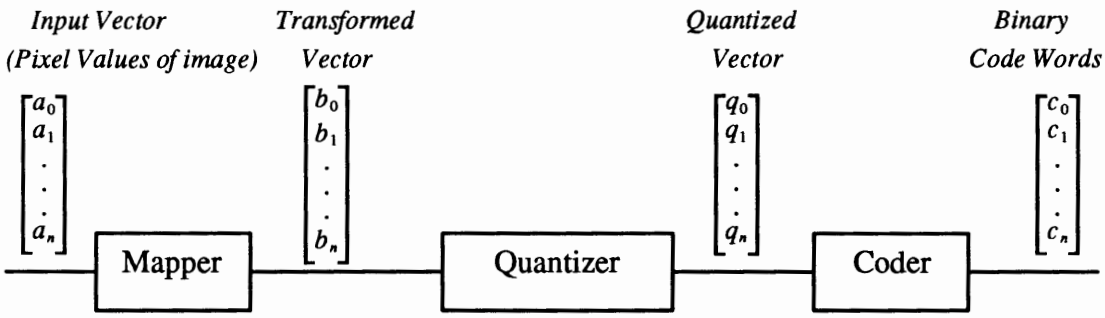


Fig. 2.1 The block diagram of the process of image compression

In the following, each of these steps will be briefly described:

2.2.1 Mapper

A mapper gets a set of image elements (pixel values) as input data and transforms it into another set of values. Depending on the type of the transformation the characteristics of the new set are different.

Some Mappers perform transformations which cause the useful information of the input data to be concentrated into a small number of samples. The Fourier transform and the Cosine transform are among those transforms used in these types of Mappers. For example in the JPEG compression method, which uses the Discrete Cosine transform (DCT) [16], the mapper causes the more useful information of the input matrix to be concentrated in the upper left corner of the transformed matrix (Fig. 2.2).

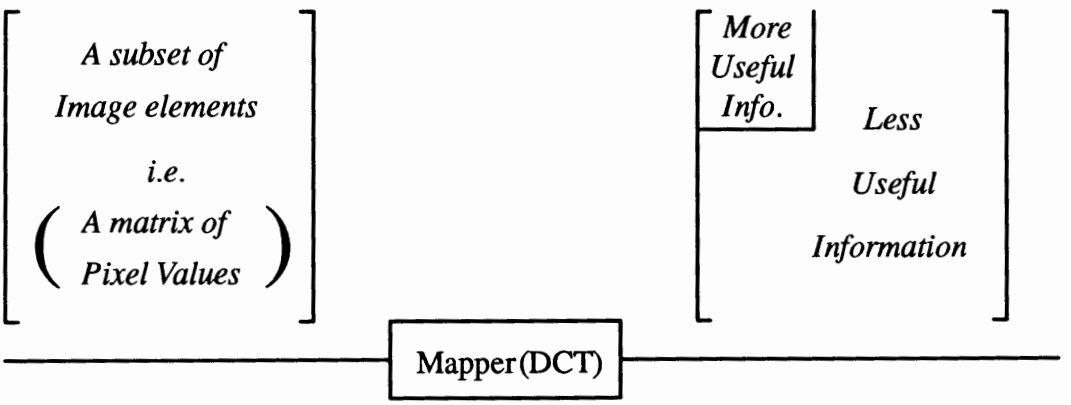


Fig. 2.2 The effect of DCT on the input matrix

If the less useful information in the output matrix is discarded, a high compression factor can be obtained.

In these transforms usually every single element of the output vector (matrix) depends on all elements of the input vector. Consequently the process is quite slow and also the hardware for these Mappers is somehow complicated.

Some other mappers transform the input data in such a way that the redundant information in the output is decreased. The transform used in the differential encoding method is a good example of this type of Mapper. In these transforms usually each element of the output vector depends on a subset of the input elements.

A linear transform can be represented in general as in Eq. (2.1)

$$\begin{bmatrix} b_0 \\ b_1 \\ \cdot \\ \cdot \\ \cdot \\ b_n \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & \cdot & \cdot & m_{1n} \\ m_{21} & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ m_{m1} & \cdot & \cdot & \cdot & m_{mn} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ a_n \end{bmatrix} \qquad (2.1)$$

In black and white images the elements of the input vector $[a_0, a_1, \ldots a_n]^T$ are usually represented by 8 bits (one byte). In some methods the input matrix is divided into 8 matrices corresponding to bit planes from the most significant bit plane to the least significant bit plane, and the transformation is performed on each bit plane separately. These transformations are binary, and they usually cause the number of nonzero elements to be decreased. The Reed–Muller transform belongs to this group. In this method the mapping process is usually performed in two steps. First, a permutation matrix is defined to permute the elements of the input bit planes, then the permuted data is transformed by the transformation matrix. In other methods such as the Haar transform, these two steps are also performed but there is no need for dividing the matrix into binary plane matrices

(Fig. 2.3). In this method the permutation is done in such a way that the transformation gives the minimum number of nonzero elements.

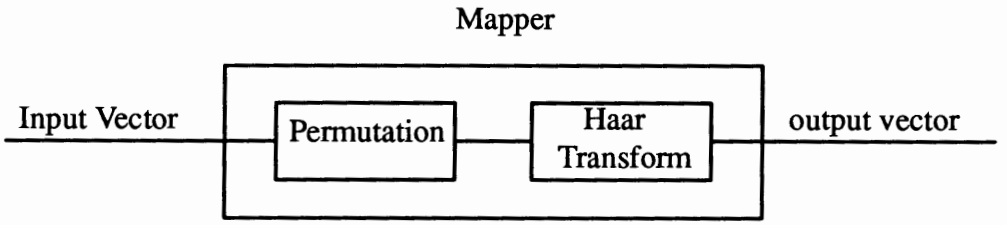


Fig. 2.3 Function of the mapper when Haar transform is used

An important point about the Mappers is that the Mapper operation is reversible. In other words the input vector can be reconstructed if the output vector is known. This comes from the fact that there is not any information loss in the operation.

2.2.2. Quantizer

In general the function of a quantizer is to take some data as the input, and generate corresponding data at the output, but the output data can have just a limited number of possible values. An example of a quantizer can be a function that generates the integer part of a real number. As it is shown in Fig. 2.4, the x variable can have any real number but the function y can have just integer numbers whose values are smaller and closest to x, i.e. $y=[x]$.

It is obvious that the operation of a quantizer is not reversible, in other words knowing the output, the input cannot be reconstructed. Therefore, in the case when error–free image compression techniques are required, a quantizer cannot be used. When some error

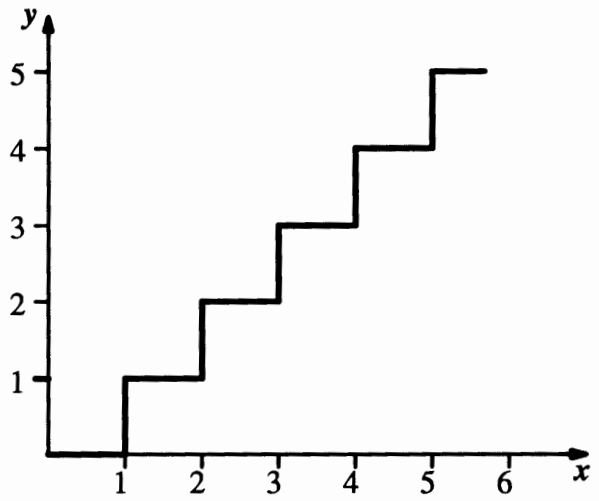


Fig. 2.4 The function for a type of quantizer

in the reconstructed image is tolerable, quantizing the data is an excellent technique to receive a high compression factor.

It is important to note that for some methods such as the JPEG algorithm, quantization of the data must be performed, otherwise no compression can be obtained. For these methods the mappers perform transformation in such a way that quantization may be exploited well.

For other methods, quantization can be utilized to obtain a better compression factor, although without quantization the compression factor still remains higher than one.

2.2.3. Coder

A coder in an image compression system takes data from the quantizer (or from the mapper when there is no any quantizer in the system) and generates the codewords which usually contain a lower number of bits to represent an image.

The operation of a coder is reversible because it assigns a unique codeword C_i to each input value q_i , so that knowing the output, the input value can be reconstructed.

Depending on the method of compression, different types of coding can be used. In some methods a specific codeword is assigned to each input value, but the length of the codewords on the average is less than the length of the input values. In chapter 3, Run–Length coding, which is the most common used type of coding used for image compression, will be explained.

Chapter 3

Run–Length Coding

3.1. Introduction

Run–length coding is a data compression method which is used to code any type of repeating character sequences. This method of coding has been widely used in several fields such as facsimile communication and image compression.

As mentioned before, in Reed–Muller image compression each picture matrix is converted to eight bit plane matrices which consist of two different values of elements, i.e. 0 and 1. Therefore to code the data of a bit plane matrix, methods similar to those used for facsimile communication can be used, because the facsimile signals also consist of two different elements (The facsimile signals obtained by scanning the document comprise just black and white picture elements). We can assume white as 0 and black as 1. In this chapter, three methods of Run–Length coding used in facsimile communication and image compression are explained:

1) Conventional Run–Length Coding (RLC)

2) Relative Address Coding (RAC)

3) Relative Element Address Designate (READ)

In each of the above methods there is a need to find Huffman codes for different 0 (white) runs or 1 (black) runs. First the algorithm to generate Huffman codes is explained.

3.2. Huffman Algorithm

Huffman coding is one of the well–known methods for effectively coding symbols. Huffman coding creates variable length codes that have an integral number of bits, and symbols with higher probability receive shorter codes. Decoding a stream of Huffman codes is generally done by following a binary decoder tree.

Using an example this algorithm is shown clearly: assume there are 5 symbols (A, B, C, D, E) in our stream with different frequencies, for example we have 16 As, 8 Bs, 6 Cs, 6 Ds and 5 Es (Fig. 3.1)

16	8	6	6	5
A	B	C	D	E

Fig. 3.1 Example of five symbols

Now using the following rules we find the Huffman codes for the symbols:

a) The two free nodes with the lowest weights are located. In this example these are E and D with weights of 5 and 6 (the tie between C and D was broken arbitrarily and it will not affect the compression ratio).

b) A parent node for these nodes is created. It is assigned a weight equal to the sum of the two child nodes (Fig. 3.2).

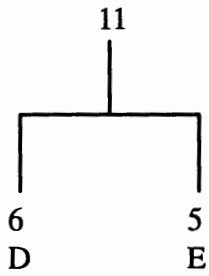


Fig. 3.2 The first step to code the symbols

c) The parent is added to the list of free nodes, and the two child nodes are removed from the list. Therefore in our example, E and D are removed from the free list.

d) One of the child nodes is designated as the path taken from the parent node when decoding a 0 bit, the other is arbitrarily set to the 1 bit. In our example D is then assigned to the 0 branch of the parent node and E is assigned to the 1 branch (Fig. 3.3). These two bits will be the LSBs of the resulting codes.

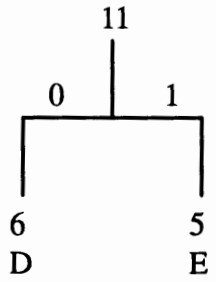


Fig. 3.3 the first bit assignment to D and E

e) The previous steps are repeated until only one free node is left. This free node is designated to become the root of the tree.

On the next pass through the list of free nodes, the B and C nodes are then taken as the two with the lowest weights. These are then attached to a new parent node, and the

parent node is assigned a weight of 14, and B and C are removed from the free node list. At this point the tree is as shown in Fig. 3.4.

On the next pass the two nodes with the lowest weights are the parent nodes for B/C and D/E pairs. These are tied together with a new parent node which is assigned a weight of 25.

Continuing the steps the final result will be as in Fig. 3.5.

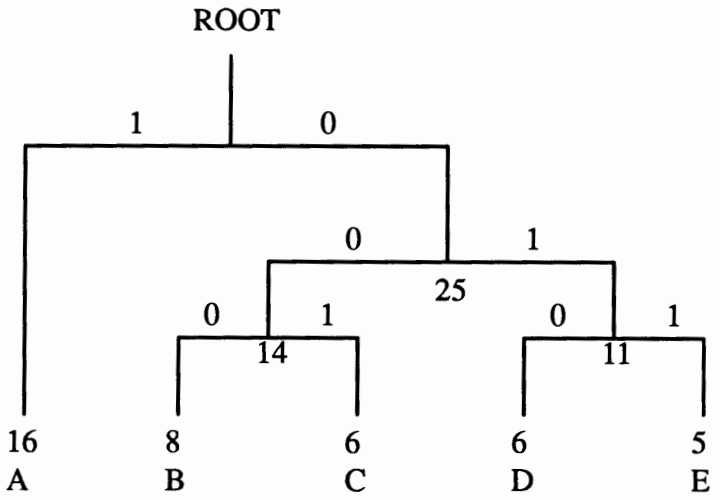


Fig. 3.5 The final Huffman tree for the symbols

From the tree the Huffman codes for our symbols will be as in Fig. 3.6.

A: 1

B: 000

C: 001

D: 010

E: 011

Fig. 3.6 The Huffman codes for the symbols

3.3. Run–Length Coding

In general there are two methods of Run–Length Coding:

1) One dimensional coding

2) Two dimensional coding

The conventional Run–Length Coding (RLC) is a one–dimensional coding method, while the Relative Address Coding (RAC) and Relative Element Address Designate are two–dimensional coding methods. In the following all three methods will be discussed:

3.3.1. One dimensional Run–Length Coding (RLC)

In this method each matrix is regarded as a sequence of alternating independent runs of 0 (white) and 1 (black) elements. So a table can be formed that consists of two columns, one containing all the 1 (black) run–length values and the other containing all the 0 (white) run–length values. Then the probability of occurrence of each run–length can be calculated from the table. Now based on these probabilities, Huffman coding can be performed. It can be shown that Huffman's procedure [x] is the optimum method of constructing a uniquely decodable and instantaneous code which has the smallest average code word length for a given independent run.

As an example, assume Fig. 3.7 shows a part of a facsimile signal from a document. The RLC code from point **A** to point **B** can be computed as follows:

Fig. 3.7 A sample of 1 and 0 runs

If we assume that the value 0 represents white elements and that the value 1 represents black elements, from **A** to **B** there are: a run of 14 white elements, then a run of 8 black elements, and then a run of 6 white elements. The Huffman code table is used for the facsimile signals which has been created based on the probability of the occurrence of different white and black runs in a great number of different documents. According to this table, the code for 14 white run is 110100, for 8 black run it is 10011 and for 6 white run it is 1110. So the resulting Run–Length code for the string from **A** to **B** is 110100100111110.

3.3.2. Compression efficiency

If each zero run–length value is shown by r_0 and the probability of its occurrence by $P(r_0)$ the average zero run–length value ($\bar{r}_0$) can be calculated by Eq. (3.1):

$$\bar{r}_0 = \sum_{r_0=0}^{n} r_0 . P(r_0) \qquad (3.1)$$

where **n** is the largest value of r_0 .

The average amount of information in bits for each Zero Run is given by the entropy $\mathbf{H_0}$, as in Eq. (3.2).

$$H_0 = - \sum_{r_0=0}^{n} P(r_0).\log_2 P(r_0) \qquad (3.2)$$

In fact $\mathbf{H_0}$ is the minimum average number of bits needed to code r_0s. Similar equations can be written for the average one (white) run–length value r_1 and the entropy of the one runs $\mathbf{H_1}$.

To find the maximum theoretical compression factor $\mathbf{Q_{max}}$ for a given set of run–length values, we see that the average number of bits in a one–run is $\bar{r}_1$ and in a zero–run is $\bar{r}_0$, while we need $\mathbf{H_1}$ number of bits to code $\bar{r}_1$ and $\mathbf{H_0}$ number of bits to code $\bar{r}_0$, so the maximum compression factor can be calculated by Eq. (3.3):

$$Q_{max} = \frac{\bar{r}_1 + \bar{r}_0}{H_1 + H_0} \qquad (3.3)$$

In Eq. (3.2), $-\log_2 P(r_0)$ shows the number of bits required to encode run–length r_0 , and is not necessarily an integer number. Huffman coding gives a way of rounding this number to a closed integer value. Because of this, if $n(r_0)$ is the length of the code word representing the zero run–length r_0 , the average number of bits representing the zero runs will be equal to or more than $\mathbf{H_0}$ and less than $\mathbf{H0+1}$ as stated in Eq. (3.4):

$$H_0 \leq \sum_{r_1=0}^{n} n(r_0).P(r_0) < H_0 + 1 \qquad (3.4)$$

Therefore the maximum compression factor as in Eq. (3.3) cannot be obtained. Another important issue in facsimile communication or in an image compression technique is that the Huffman codes are not defined specifically for every single document or matrix. In fact Huffman codes are defined based on statistics averaged over many typical documents. So these codes usually are not optimum for a specific matrix, and this causes the compression factor to be decreased.

3.3.3. Relative Address Coding (RAC)

Relative Address Coding (RAC) is a two dimensional coding method which was first used to code facsimile signals. In conventional Run Length Coding (RLC) the address of every kind of transition element is represented by the distance from the preceding transition element in the same line in terms of the number of picture elements, making a good use of the statistical intra–line correlation. Relative Address Coding not only has taken advantage of the intra–line correlation but also exploited the inter–line correlation of the signals of the picture. So RAC can cause a remarkable reduction of redundant information in the data. Since each matrix plane consists of two kinds of information (0&1) and the information of each line is highly correlated to that of its adjacent lines, RAC can be a very useful coding method for this type of case. Before explaining the principle of RAC, some important terms need to be defined.

3.3.3.1. Transition Elements

As it has been described before, to reconstruct a matrix plane at the receiving end, it is sufficient to encode and transmit the addresses of only the transition elements (a transition element is the element which is different from its previous element in the same line). In RAC the transition elements are classified into three types, as shown in Fig. 3.8.

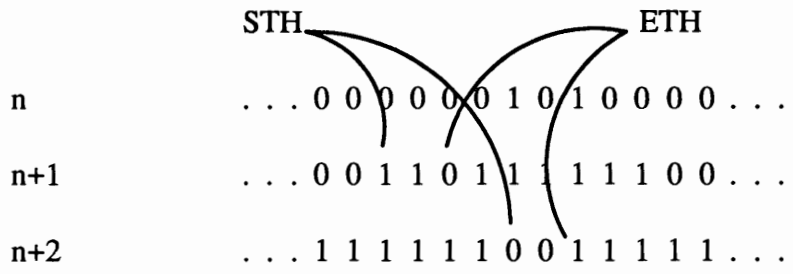


Fig. 3.8 STHs and ETHs in a sample of 0 and 1 runs

I) Starting Transition element of Head run (STH):

A head run is a One Run or a Zero Run that there exists no elements on the preceding line which are of the same type and are adjacent to the run. An STH is the first element of a head run.

For example Fig. 3.8 shows a part of a plane matrix. In this figure in Row number n+1 there are 3 Zero Runs and 2 One Runs. The first One Run of this line is a head run because a 1 does not exist in line number n, i.e. preceding line, adjacent to this run. There-

fore, the first element of this run is an STH. While the other runs in this line are not head run. Another head run is the Zero Run in Row number n+2, because there is no zero in Row number n+1 adjacent to this run.

II) Ending Transition element of a Head run (ETH):

An ETH is the transition element next to the head run. For example in Fig. 3.6 we have 2 of ETHs because we have 2 head runs.

III) Displacement Transition element (DTE):

A DTE is a transition element which is neither STH nor ETH. For example in Fig. 3.8 there are 2 STHs, 2 ETHs and the rest of the transition elements are DTEs.

3.3.3.2. Principles of the RAC method

Fig. 3.9 shows a part of a plane matrix. Using this example the principle of RAC method will be shown.

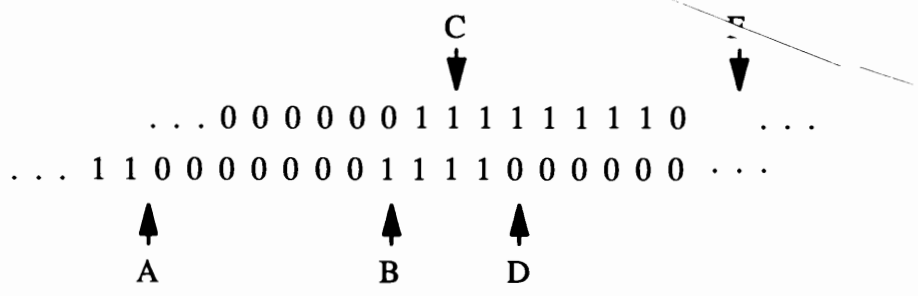


Fig. 3.9 Transition elements in a part of a plane matrix

In Fig. 3.9 the transition element **B** is shown. In regard to the transition element **B**, two reference elements are selected according to the following rules:

a) The first reference element **A** is the preceding transition element on the same line to the transition element **B**. In the case that the first reference element defined above does not exist, the first element on the line is to be the first reference element.

b) The second reference element **C** is the transition element that has the same direction of transition as the transition element **B** and is the nearest to the first reference element **A** on its right side. In the case that the element defined above does not exist, the second reference element is the imaginary element next to the last element on the preceding line.

The address of transition element **B** is encoded by the distance from the standard element that is selected between the two reference elements **A** and **B** according to the following rules:

a) If the transition element is an STH, the standard element is **A**.

b) If the transition element is an ETH, the standard element is **A**.

c) If the transition element is a DTE, in the case that the distance from the first reference element **A** to the transition element **B** is more than one element and that the first reference element **A** is nearer to **B** than the second reference element **C**, the first reference element **A** is selected to be the standard element, and the distance is expressed by the number without sign. In all other cases, the second reference element **C** is selected to be the standard element, and the distance from **C** to **B** is expressed by the number with "+" if **C** is just upon or to the left of **B**, and is expressed by the number with "–" if **C** is to the

right of **B**. Table 3.1 shows the summary of this sign assignments.

position of the reference element in relation to the transition element	sign
On the same line	No sign
On the preceding line, just upon or left	+
On the preceding line, right	–

Table 3.1 Sign assignment for relative distances

For example in Fig. 3.9 in the case of coding the transition element **B**, since the distance **AB**(=7) is greater than the distance **BC**(=1), **C** is selected to be the standard element, and since **C** is on the right of **B** the address of **B** is encoded with a "–" sign. As a result the number used for the address of **B** is "–1". In a similar manner it can be shown that the address of transition element **D** will be "4" because the distance **DB**(=4) is smaller than **DE**(=5) and **B** is also on the same line as **D**, so there is no sign for that.

To encode the address of each transition element Huffman codes can be used. By finding the probability of the occurrence of every distance to be coded by computer simulation, it is possible to find the best Huffman code for this method.

3.3.4. Relative element address designate (READ)

READ is a two dimensional coding technique which is the most efficient method to code the facsimile signals. In this section this method is explained and later its efficiency

for bit plane matrices will be discussed .

3.3.4.1. Transition elements

In READ, the transition elements are classified into five types. Taking the example given in Fig. 3.10, these elements are explained.

$\mathbf{a_0}$: The transition element on the coding line whose position is defined by the previous coding mode which is described later. This element is the reference element.

$\mathbf{a_1}$: The next transition element on the coding line to the right of $\mathbf{a_0}$.

$\mathbf{a_2}$: The next transition element on the coding line to the right of $\mathbf{a_1}$.

$\mathbf{b_1}$: The first transition element on the reference line to the right of $\mathbf{a_0}$ whose color is opposite to $\mathbf{a_0}$.

$\mathbf{b_2}$: The next transition element on the reference line to the right of $\mathbf{b_1}$.

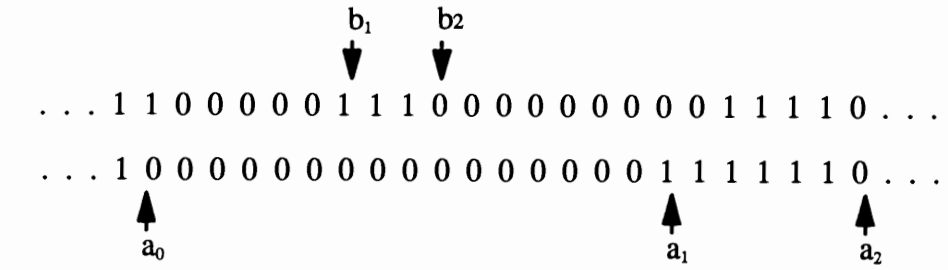
If any of the above coding elements are not detected at any time during the coding of the line, then they are set on an imaginary element positioned just after the last actual element.

4.3.4.2. Coding Modes

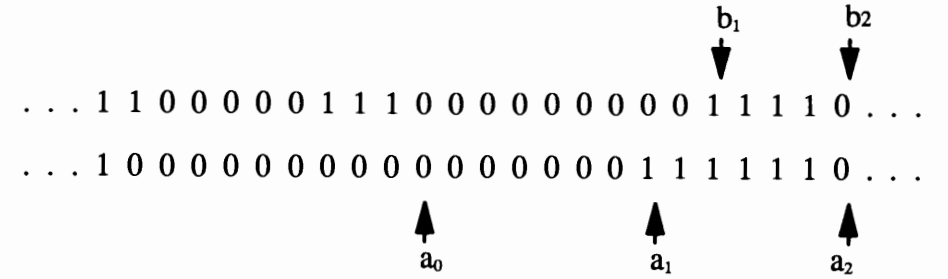
There are three kinds of coding modes defined for READ:

1) Pass Mode: As shown in Fig. 3.10 (a), the state where $\mathbf{b_2}$ lies to the left of $\mathbf{a_1}$ is defined as the Pass Mode. However when the position of $\mathbf{b_2}$ is just upon $\mathbf{a_1}$, it is not regarded as the Pass Mode.

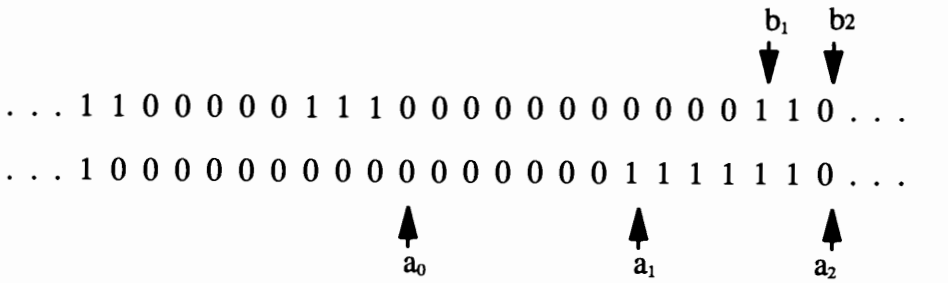
2) Vertical Mode: In this mode the position of $\mathbf{a_1}$ is coded relative to the position of $\mathbf{b_1}$. The relative distance $\mathbf{a_1b_1}$ is equal to or less than three elements, so it can assume one of



(a) Pass Mode



(b) Vertical Mode



(c) Horizontal Mode

Fig. 3.10 Examples of Pass Mode, Vertical Mode and Horizontal Mode

the seven values **V(0)**, $\mathbf{V_r(1)}$, $\mathbf{V_r(2)}$, $\mathbf{V_r(3)}$, $\mathbf{V_l(1)}$, $\mathbf{V_l(2)}$, $\mathbf{V_l(3)}$ each of which is represented by a separate codeword. The subscripts **r** and **l** indicate that $\mathbf{a_1}$ is to the right or

left of $\mathbf{b_1}$ respectively. The number in brackets shows the value of the distance $\mathbf{a_1b_1}$. For example in Fig. 3.10 (b) $\mathbf{a_1b_1}$ is coded by codeword used for $\mathbf{V_l(2)}$.

3)Horizontal Mode: If the Pass mode and the Vertical mode cannot be used to code the position of $\mathbf{a_1}$, then the Horizontal mode coding is used, Fig. 3.10 (c). In this method the run–lengths $\mathbf{a_0a_1}$ and $\mathbf{a_1a_2}$ are coded.

3.3.4.3. The coding procedure

When one of the Pass, Vertical, and Horizontal modes is detected, codes based on the following are generated.

–If Pass Mode is detected, i.e. $\mathbf{b_2}$ is detected before $\mathbf{a_1}$, then it is coded by pass mode code'0001'. Then the reference element $\mathbf{a_0}$ is set on the element just below $\mathbf{b_2}$ as the new starting element for the next coding.

–If Pass Mode is not detected, two cases are possible:

a) If $|\mathbf{a_1b_1}| \leq 3$ then vertical mode coding is selected, and $\mathbf{a_0}$ is set on the position of $\mathbf{a_1}$ for the next coding procedure. The codes for this case are the following :

V(0): 1

$\mathbf{V_r(1)}$: 011

$\mathbf{V_r(2)}$: 000011

$\mathbf{V_r(3)}$: 0000011

$\mathbf{V_l(1)}$: 010

$\mathbf{V_l(2)}$: 000010

$\mathbf{V_{l(3)}}$: 0000010

b) If $|\mathbf{a_1b_1}| > 3$ then Horizontal mode coding is selected, and positions of $\mathbf{a_1}$ and $\mathbf{a_2}$ are coded. $\mathbf{a_2}$ is then regarded as the new position of the reference element $\mathbf{a_0}$. The code-word to code $\mathbf{a_1}$ and $\mathbf{a_2}$ is found from the term $\mathbf{H+M(a_0a_1)+M(a_1a_2)}$. In this term **H** is coded as '001', and $\mathbf{M(a_0a_1)}$ and $\mathbf{M(a_1a_2)}$ are defined by Huffman code tables that have been generated based on the probability of the occurrence of those runs in the picture.

An important point about two–dimensional coding methods is that since the coding of each transition element depends on the elements in the preceding line, if an error occurs in one place it could cause errors in the following lines. In order to prevent this undesirable situation, a mixing of two–dimensional coding and one dimensional coding is used. Usually after one line is coded one–dimensionally, K–1 successive lines are coded two–dimensionally. In READ coding, the value of K is usually 2 or 4.

Chapter 4

JPEG Algorithm for lossy image compression

4.1. Introduction

In this chapter the steps involved in the JPEG algorithm [15],[16] for lossy image compression are explained. This method is the best existing compression method for continuous–tone still images from the point of view of compression and quality factors. But this process requires a large number of summations and multiplications which make the process slow. Therefore if the speed is not important for us this method is the best.

4.2. Compression Algorithm

In JPEG algorithm for lossy image compression each image is considered as a matrix of pixel values. Then this matrix is divided to submatrices of size 8×8, and on each submatrix the steps shown in Fig. 4.1 are applied.

4.2.1. The Discrete Cosine Transform (DCT):

The key to the compression process based on JPEG is the Discrete Cosine Transform (DCT). The DCT is one of the mathematical transforms that includes Fast Fourier Transform (FFT). The basic operation of these transforms is to take a signal and transform it from one type of representation into another. In an electric signal each point of the signal shows the amplitude (e.g. the voltage level) of the signal in time domain. The FFT

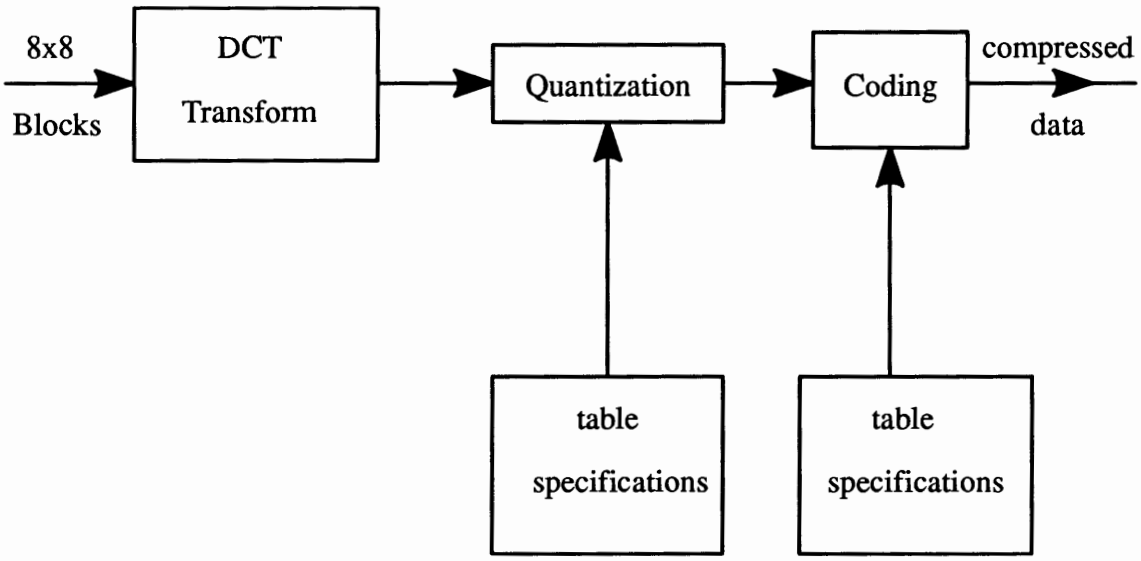


Fig. 4.1 The block diagram of JPEG process for lossy compression

transforms this signal into a set of frequency values that describes exactly the same signal.

The DCT is closely related to the Fourier Transform. It takes a set of points from the spatial domain and transforms them into an identical representation in the frequency domain. In our case the signal is a graphical image, so instead of a two–dimensional signal plotted on the X and Y axis, the DCT will operate on a three dimensional signal. In fact, in this case X and Y axes are the two dimensions of the screen and the amplitude (Z axis) of the signal is the value of the pixel at a particular point on the screen. So the value on the Z axis denotes the color on the screen. In the case of black and white images the value of each pixel can vary in the range of 0 to 255 because we assign 8 bits (1 byte) for each pixel, and in the case of color images we assign 8 bits for each of red, green and blue colors so 24 bits are assigned for each pixel.

The formula for the two dimensional DCT is as in Eq. (4.1).

$$DCT(i,j) = \frac{1}{\sqrt{2N}} C(i).C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} Pixel(x,y).cos\frac{(2x+1)i\Pi}{2N} cos\frac{(2y+1)j\Pi}{2N} \quad (4.1)$$

$$C(x) = \frac{1}{\sqrt{2}} \qquad if \quad x = 0$$

$$C(x) = 1 \qquad if \quad x > 0$$

Where *Pixel(x,y)* represents the amplitude (intensity) of the pixel at point (x,y). In other words *Pixel(x,y)* is the value of the element in the *x*th column and *y*th row of the 8×8 block (*Pixel matrix).* Eq. (4.1) is the mathematical definition of the N×N DCT. Since in the case of JPEG each block is 8x8, the equation will be as in Eq. (4.2):

$$DCT(i,j) = \frac{1}{4} C(i).C(j) \sum_{x=0}^{7} \sum_{y=0}^{7} Pixel(x,y).cos\frac{(2x+1)i\Pi}{16} cos\frac{(2y+1)j\Pi}{16} \quad (4.2)$$

$$C(x) = \frac{1}{\sqrt{2}} \qquad if \quad x = 0$$

$$C(x) = 1 \qquad if \quad x > 0$$

As an example if the input consists of an 8×8 matrix of pixel values as in Fig. 4.2, the DCT matrix using equation (4.2) will be as in Fig. 4.3.

$$Pixel = \begin{bmatrix} 139 & 144 & 149 & 153 & 155 & 155 & 155 & 155 \\ 144 & 151 & 153 & 156 & 159 & 156 & 156 & 156 \\ 150 & 155 & 160 & 163 & 158 & 156 & 156 & 156 \\ 159 & 161 & 162 & 160 & 160 & 159 & 159 & 159 \\ 159 & 160 & 161 & 162 & 162 & 155 & 155 & 155 \\ 161 & 161 & 161 & 161 & 160 & 157 & 157 & 157 \\ 162 & 162 & 161 & 163 & 162 & 157 & 157 & 157 \\ 162 & 162 & 161 & 161 & 163 & 158 & 158 & 158 \end{bmatrix}$$

Fig. 4.2 An example of 8×8 matrix of pixel values

$$DCT_{pixel} = \begin{bmatrix} 235.6 & -1 & -12.1 & -5.2 & 2.1 & -1.7 & -2.7 & 1.3 \\ -22.6 & -17.5 & -6.2 & -3.2 & -2.9 & -0.1 & 0.4 & -1.2 \\ -10.9 & -9.3 & -1.6 & 1.5 & 0.2 & -0.9 & -0.6 & -0.1 \\ -7.1 & -1.9 & 0.2 & 1.5 & 0.9 & -0.1 & 0 & 0.3 \\ -0.6 & -0.8 & 1.5 & 1.6 & -0.1 & -0.7 & 0.6 & 1.3 \\ 1.8 & -0.2 & 1.6 & -0.3 & -0.8 & 1.5 & 1.0 & -1.0 \\ -1.3 & -0.4 & -0.3 & -1.5 & -0.5 & 1.7 & 1.1 & -0.8 \\ -2.6 & 1.6 & -3.8 & -1.8 & 1.9 & 1.2 & -0.6 & -0.4 \end{bmatrix}$$

Fig. 4.3 DCT matrix of the example

The DCT matrix shows the spectral compression characteristics. The position (0,0) in the upper left–hand corner of the matrix shows the " DC coefficient " which in our example is 235.6. This value represents an average of the overall magnitude of the input matrix. We should note that the DC coefficient is almost an order of magnitude greater than any of the other values in the DCT matrix, and also as the elements move farther and farther from the DC coefficient, they tend to become lower and lower in magnitude. This means that by performing the DCT on the input data, we have concentrated the representation of the image in the upper left coefficients of the DCT matrix, with the lower right coefficients of the DCT matrix containing less useful information.

To reconstruct the image sample from the DCT matrix, we can use the formula for Inverse Discrete Cosine Transform (IDCT) which is given in Eq. (4.3).

$$Pixel(x,y) = \frac{1}{\sqrt{2N}} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} C(i).C(j).DCT(i,j).cos\frac{(2x+1)i\Pi}{2N} cos\frac{(2y+1)j\Pi}{2N} \qquad (4.3)$$

$$C(x) = \frac{1}{\sqrt{2}} \qquad if \quad x = 0$$

$$C(x) = 1 \qquad if \quad x > 0$$

And for the case of JPEG the equation will be as in Eq. (4.4):

$$Pixel(x,y) = \frac{1}{4} \sum_{i=0}^{7} \sum_{j=0}^{7} C(i).C(j).DCT(i,j).cos\frac{(2x+1)i\Pi}{16} cos\frac{(2y+1)j\Pi}{16} \qquad (4.3)$$

$$C(x) = \frac{1}{\sqrt{2}} \qquad if \quad x = 0$$

$$C(x) = 1 \qquad if \quad x > 0$$

From Eq. (4.2) we see that the creation of DCT matrix is straightforward and actually is a doubly nested loop. The C code to create the DCT should be something as:

```
for( i=0 ; i<8 ; i++ )
  for( j=0 ; j<8 ; j++) {
      temp=0;
      for( x=0 ; x<8 ; x++ )
        for( y=0 ; y<8 ; y++ )
```

```
            temp += pixel[x][y] * cosine[x][i] * cosine[y][j];
        temp *= (1/4) * coefficient[i][j];
        DCT[i][j]=temp;
    }
```

One can observe that the inner element of the loop gets executed 8*8=64 times for every DCT element that is calculated.

A more efficient method to create DCT matrix is to use Cosine Transform Matrix, C=[$C_{i,j}$], which is defined as follows:

$$C_{i,j} = \begin{cases} \frac{1}{\sqrt{N}} & if \quad i = 0 \\ \sqrt{\frac{2}{N}} Cos\left[\frac{(2j+1)i\pi}{2N}\right] & if \quad i > 0 \end{cases} \tag{4.5}$$

and the DCT matrix can be calculated as follows:

$$DCT = C * Pixel * C^T \tag{4.6}$$

which C^T is the C transpose, and * operator refers to matrix multiplication. If we do this method for pixel matrix of Fig. 4.2, the result will be as in Fig. 4.3.

4.2.2. Quantization

The goal of this processing step is to discard information which is not visually significant. Quantization is defined as the division of each DCT coefficient by its corresponding

quantizer step size in the Quantization Matrix, followed by rounding to the nearest integer:

$$Quantized\ Value(i,j) = \frac{DCT(i,j)}{Quantizaion\ Matrix(i,j)}\ rounded\ to\ the\ nearest\ integer \quad (4.7)$$

Each element of the Quantization Matrix can be any integer value from 1 to 255 which specifies the step size of the quantizer for its corresponding DCT coefficient.

Since the quantization matrix can be defined at runtime when the compression takes place, JPEG allows for the use of any quantization matrix; however, the creators of JPEG have developed a standard set of quantization values supplied for use by the implementers of the JPEG code. These tables are based on extensive testing by members of the JPEG committee and they provide a good baseline for levels of compression. It is obvious that if we choose high step sizes for most DCT coefficients, we will obtain excellent compression ratios and poor picture quality. Conversely if we choose low step sizes, the compression ratios would not be very good, but the picture quality should be excellent. And based on these step sizes, the quality factor of the process is defined which can be up to 100. If the quality factor is 100 it means all of the elements of quantization matrix are 1 and in fact the reconstructed image will be the same as the original, so that we do not gain anything from the process. For the quality factors less than 25, although the compression ratio is excellent, the picture quality has degraded far enough to make further degradation of the quality factor unacceptable.

Fig. 4.4 shows the example quantization table for gray scale components included in the informational annex of the draft JPEG standard.

If we quantize the DCT matrix from Fig. 4.3 by the quantization matrix from Fig. 4.4 using Eq. (4.7), the quantized matrix will be as in Fig. 4.5.

$$Quantized\ Matrix = \begin{bmatrix} 15 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Fig. 4.5 The quantized matrix of the example

It can be seen that most of the elements in the quantized matrix are Zero so the data from this matrix can be compressed by the entropy encoding process which will be discussed in the next section.

4.2.3. Coding

Coding the quantized images is the final step in the JPEG process. This step itself consists of three steps.

The first step is the DC coding, in fact the DC coefficient is treated separately from the 63 AC coefficients. Because there is usually a strong correlation between the DC coefficients of adjacent 8x8 blocks, the quantized DC coefficient is encoded as the difference from the DCT term of the previous block in the encoding order. In our example (Fig. 4.5) the DC component is 15, and if the DC component of the previous block is for exam-

ple 12, then 15–12=3 will be the DC component of our new code.

$$\begin{bmatrix} DC_{i-1} \\ \\ Block(i-1) \\ \\ \end{bmatrix} \quad \begin{bmatrix} DC_i \\ \\ Block(i) \\ \\ \end{bmatrix} \quad DC = DC_i - DC_{i-1}$$

Fig. 4.6 Determination of the DC component of the new matrix

The second step is that all of the quantized coefficients are ordered into the zig–zag sequence. Since in the quantized matrix so many values are set to zero (Fig. 4.5), the JPEG committee decided to handle zero values differently from other coefficient values. In fact for zero values they use Run–Length Encoding (RLE) algorithm, while for other values they use Huffman or arithmetic coding.

In Run–Length Encoding a simple code is developed that gives a count of consecutive zero values in the image. Since over half of the coefficients are quantized to zero in many images, this gives an opportunity for excellent compression.

When the coefficients in the zig–zag sequence are reordered, it increases the length of runs so the compression ratio increases. In the zig–zag sequence the JPEG algorithm moves through the block along diagonal paths selecting what should be the highest value elements first, and working its way toward the values which are likely to be the lowest. Fig. 4.7 shows the zig–zag sequence for our example.

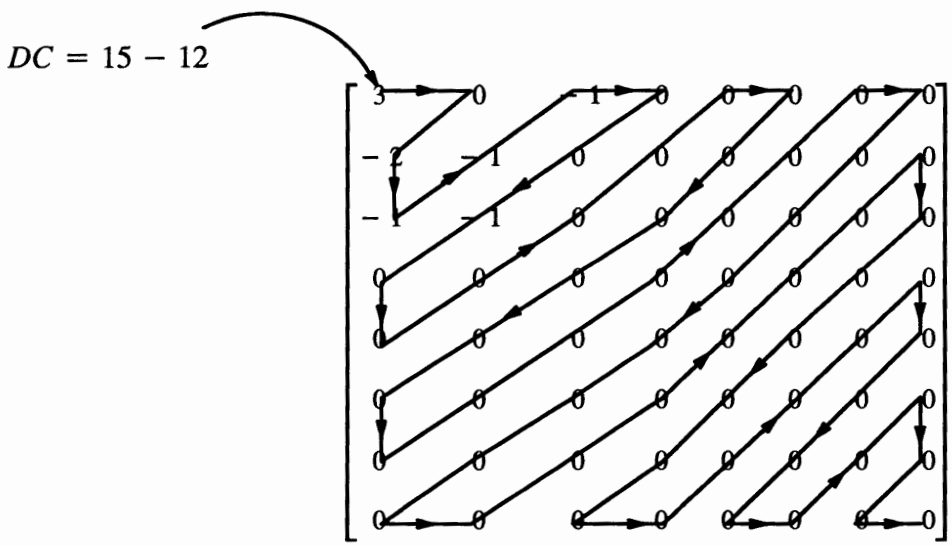


Fig. 4.7 Moving through the matrix in zig–zag form

Now we continue the coding process of JPEG algorithm by introducing one of the JPEG standards after the zig–zag sequence step. This method is called "Baseline Sequential Coding", and in this method the Huffman coding will be used. The first step is to represent some symbols for sequences of values that exist after the zig–zag sequence. These symbols are elements of what is called the Intermediate Symbol Sequence. In the Intermediate Symbol Sequence, each nonzero AC coefficient is represented in combination with the runlength (consecutive number) of zero–valued AC coefficients which precede it in the zig–zag sequence. Each runlength / nonzero–coefficient combination is represented by a pair of symbols:

Symbol 1	Symbol 2
(Runlength , Size)	(Amplitude)

In Symbol 1 "Runlength" is the number of consecutive zero–valued AC coefficients in the zig–zag sequence preceding the nonzero AC coefficient which is presented. "Size" is the number of bits used to encode the "Amplitude", and "Amplitude" is simply the amplitude of the nonzero AC coefficient.

For our example (Fig. 4.7) we can find the symbols as follows:
The first number of the block (3) is the DC term which must be encoded differently. For this term the Intermediate representation is (2)(3) because the amplitude is 3 which requires 2 bits (Size). (See Table 4.1).

Next the quantized AC coefficients are encoded. If we follow the zig–zag order, the first nonzero coefficient is –2 preceded by a Zero Run of 1, so the intermediate symbol for this term is (1,2)(–2).

Next encountered in the zig–zag order are three consecutive nonzeros of amplitude –1. This means each is preceded by a zero–run of length zero, so the intermediate symbols are (0,1)(–1). The last nonzero coefficient is –1 preceded by two zeros so the symbol is (2,1)(–1). Since this is the last non–zero coefficient, the final symbol should represent the End Of Block (EOB) which is (0,0)

The intermediate symbol sequence for our example is then as follows:
(2)(3), (1,2)(–2), (0,1)(–1), (0,1)(–1), (0,1)(–1), (2,1)(–1), (0,0)

Now to encode the intermediate symbols we use the following rules:
a) Each Symbol 1 is encoded with a Variable–Length Code (VLC) from the Huffman

Size	Amplitude
1	–1 , 1
2	–3,–2 , 2,3
3	–7.....–4 , 4.....7
4	–15.....–8 , 8.....15
5	–31.....–16 , 16.....31
6	–63.....–32 , 32.....63
7	–127.....–64 , 64.....127
8	–255.....–128 , 128.....255
9	–511.....–256 , 256.....511
10	–1023.....–512 , 512.....1023

Table 4.1 The size of different amplitudes

table set assigned to the 8x8 block's image component. For our example the codes will be as the following:

The differential–DC VLC is:

(2): 011

The AC luminance VLCs are:

(0,0): 1010

(0,1): 00

(1,2): 11011

(2,1): 11100

b) Each Symbol 2 is encoded with a Variable–Length Integer (VLI) code whose length in bits is given in Table 1. The codes for symbol 2 for our example are then the following:

(3): 11

(–2): 01

(–1): 0

The bit stream for given 8x8 block (Fig. 4.2) will be then as follows:

01111110110100000000001110001010

which is 31 bits, while the 8x8 block consists of 8*8*8=512 bits.
The compression ratio for this example is then 512/31 or about 16.5, which is significantly high.

This was the final step of the process and after this the compressed data will be transmitted.

4.3. Reconstruction of the block at the receiver

Entropy decoding process is the first step which is performed in the receiver. When the transmitted stream of data is received, it can easily be decoded by the Huffman code table at the receiver which is exactly the same as that in the transmitter. For example when the above stream is received, the first thing coded is 011, which according to the Huffman table corresponds to the differential–DC VLC of (2). So it realizes that the next two bits represent the value of differential–DC, and since the next two bits are 11 it knows from

another table that the value must be (3). The next thing coded is 11011 which corresponds to AC VLC of (1,2), so the decoder knows that there must be a Zero Run of length one before the nonzero element is decoded, and the nonzero element has to be 2 bits long. So it checks the next two bits which are 01 and according to the table it corresponds to the value of (–2). In a similar way all of the bits in the stream are decoded and finally after DC decoding a matrix exactly the same as the quantized matrix is reconstructed as in Fig. 4.8:

$$\textit{Reconstructed Quantized Matrix} = \begin{bmatrix} 15 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Fig. 4.8 The Reconstructed Quantized Matrix

The next step is to dequantize the *Reconstructed Quantized Matrix* and for the dequantization the following formula is used:

$$DCT(i,j) = Quantized\ Matrix(i,j) * Quantization\ Matrix(i,j) \tag{4.8}$$

The matrix of Fig. 4.8 after dequantization using Eq. (4.8) gives then the DCT matrix as in Fig. 4.9.

$$Reconstructed\ DCT_{pixels} = \begin{bmatrix} 240 & 0 & -10 & 0 & 0 & 0 & 0 & 0 \\ -24 & -12 & 0 & 0 & 0 & 0 & 0 & 0 \\ -14 & -13 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Fig. 4.9 The Reconstructed DCT Matrix

Now to reconstruct the image sample from Fig. 4.9, Eq. (4.4) is used, which is the equation for IDCT, and the result will be as in Fig. 4.10.

$$Reconstructed\ Pixels = \begin{bmatrix} 144 & 146 & 149 & 152 & 154 & 156 & 156 & 156 \\ 148 & 150 & 152 & 154 & 156 & 156 & 156 & 156 \\ 155 & 156 & 157 & 158 & 158 & 157 & 156 & 155 \\ 160 & 161 & 161 & 162 & 161 & 159 & 157 & 155 \\ 163 & 163 & 164 & 163 & 162 & 160 & 158 & 156 \\ 163 & 164 & 164 & 164 & 162 & 160 & 158 & 157 \\ 160 & 161 & 162 & 162 & 162 & 161 & 159 & 158 \\ 158 & 159 & 161 & 161 & 162 & 161 & 159 & 158 \end{bmatrix}$$

Fig. 4.10 Reconstructed Pixel Matrix

Now if one compares the reconstructed sample values (Fig. 4.10) with the original image sample (Fig. 4.2) a remarkable similarity can be observed, while the number of

bits transmitted from transmitter to receiver is much smaller than the number of bits needed to transmit an un–decoded block of pixels.

Therefore it can be seen that the JPEG algorithm is an excellent method to compress images. The only disadvantage of this method is that the process of compression is slow. It is because of the fact that the number of operations required to compress the image is very large.

Chapter 5

Image compression using Haar transform

5.1. Introduction

Spectral methods are one of the methods to analyze logical functions and other discrete mappings. Walsh and Haar functions are two important functions which have already been used for this purpose. An advantage of the Haar function, as will be explained in detail, is that the expansion coefficients of a logical function in Haar series depend on the local behavior of the function. Using this fact, Karpovsky [6] has shown that such an order of arguments can be found that gives the minimum number of nonvanishing coefficients. Image compression based on Haar function takes advantage of this minimization. In fact, the pixel values of an image are reordered in such a way that the number of nonvanishing coefficients in their Haar series gets minimized. In this chapter the Haar method will be clearly explained.

5.2. Orthogonal Representation of Logical Functions

Since Boolean functions are the only type of functions dealt with in this thesis, only this particular case is investigated. A combinational circuit with m inputs and k outputs (Fig. 5.1) can be described by Eq. (5.1):

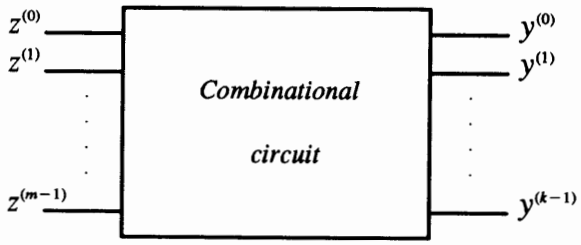


Fig. 5.1 A combinational circuit in general

$$y^{(s)} = f^{(s)}(z^{(0)}, z^{(1)}, \ldots, z^{(m-1)}) \quad s = 0, 1, \ldots, k-1 \; ; \; y^{(s)}, z^{(s)} \in \{0, 1\} \tag{5.1}$$

The logical function shown by Eq. (5.1) can also be described by a discrete function asin Eq. (5.2):

$$y = f(z) \tag{5.2}$$

where

$$z = \sum_{s=0}^{m-1} z^{(s)} 2^{m-1-s} \qquad z \in [0, 2^m) \tag{5.3}$$

$$y = \sum_{s=0}^{k-1} y^{(s)} 2^{k-1-s} \tag{5.4}$$

Now a step function $\Phi(z)$ of a real argument, defined on a half–open plane interval $[0,2^m)$, can represent our logical function as in Eq. (5.5):

$$\Phi(z) = f(\delta) \qquad z \in [\delta, \delta + 1) \tag{5.5}$$

All systems of logical functions can be analyzed by their step function representations.

To make the above notations more clear, let us consider a system described by the following Boolean functions.

$$y^{(0)} = z^{(0)} \oplus z^{(1)}$$

$$y^{(1)} = z^{(0)} \oplus z^{(2)}$$

Table 5.1 describes this function, and Fig. 5.1 shows the corresponding step function $\Phi(z)$.

$z^{(0)}$	$z^{(1)}$	$z^{(2)}$	$y^{(0)}$	$y^{(1)}$	z	$y = f(z)$
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	1	1	2	3
0	1	1	1	0	3	2
1	0	0	1	0	4	2
1	0	1	1	1	5	3
1	1	0	0	1	6	1
1	1	1	0	0	7	0

Table 5.1 Truth table of the function

The step function $\Phi(z)$ can be expanded as an orthogonal series or Fourier series as in Eq. (5.6):

$$\Phi(z) = \sum_{\omega=0}^{\infty} S(\omega)\Psi_{\omega}(z) \tag{5.6}$$

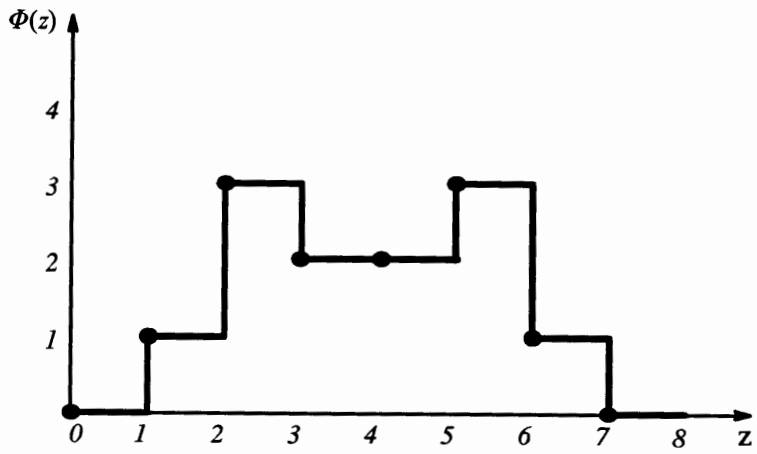


Fig. 5.2 The step function of the example

where $\Psi_{\omega}(z)$ is a complete system of orthogonal step functions defined on $[0,2^m)$, i.e.:

$$\left(\int_0^{2^m} \Psi_{\omega}(z)\Psi_{\omega}^{*}(z)dz\right)^{-1}\left(\int_0^{2^m} \Psi_{\omega}(z)\Psi_{r}^{*}(z)dz\right) = \begin{cases} 1 & if \quad \omega = r \\ 0 & if \quad \omega \neq r \end{cases} \tag{5.7}$$

where $\Psi^{*}(z)$ is the complex conjugate to $\Psi(z)$.

The coefficients S(ω) are the Fourier coefficients defined as in Eq. (5.8)

$$S(\omega) = \left(\int_0^{2^m} \Psi_{\omega}(z)\Psi_{\omega}^{*}(z)dz\right)^{-1} \int_0^{2^m} \Psi_{\omega}(z)\Psi_{\omega}^{*}(z)dz \tag{5.8}$$

Sequence of S(0), S(1), . . . is called the spectrum of the system relative to $\Psi_\omega(z)$. It can be seen that there is a one to one correspondence between the original logical function, Eq. (5.1), and its spectrum S(ω). So the original function may be analyzed by finding the spectrum of the system. This method is called a spectral method.

5.3. Haar Transform

The set of Haar functions is a complete orthogonal system defined as follows:

$$H_0^{(0)} \equiv 1$$

$$H_l^{(q)} = \begin{cases} 1 & \text{if} \quad z \in [(2q-2)2^{m-l-1}, (2q-1)2^{m-l-1}) \\ -1 & \text{if} \quad z \in [(2q-1)2^{m-l-1}, 2q.2^{m-l-1}) \\ 0 & \text{at other points of } [0, 2^m) \end{cases} \tag{5.9}$$

$$l = 0, 1, \ldots, m-1 \quad ; \quad q = 1, 2, \ldots, 2^l$$

Fig. 5.3 shows the the Haar functions for m=3.

Any system of Boolean function of *m* arguments represented by a step function Φ(z), can be shown by the Haar series as follows:

$$\Phi(z) = S_0^{(0)} H_0^{(0)}(z) + \sum_{l=0}^{m-1} \sum_{q=1}^{2^l} S_l^{(q)} H_l^{(q)}(z) \tag{5.10}$$

where $S_l^{(q)}$ is the spectrum of the function defined below:

$$S_{(l)}^{(q)} = 2^{-m+l} \sum_{z=0}^{2^m-1} \Phi(z) H_l^{(q)}(z) \tag{5.11}$$

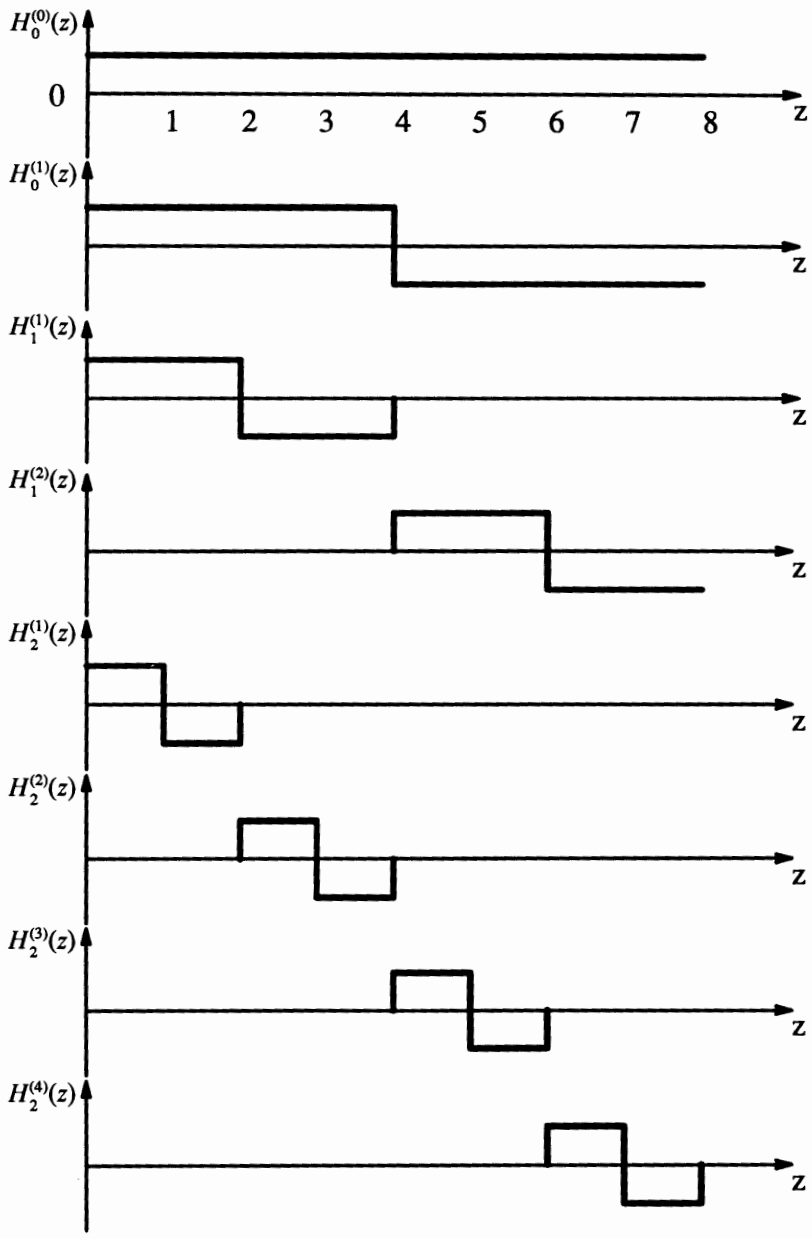


Fig. 5.3 The Haar functions for m=3

From equations (3.11) and (3.9) it can be observed that:

$$S_{m-1}^{(q)} = 2^{-1}(\Phi(2q-2) - \Phi(2q-1)) \qquad (q = 1,2,\ldots,2^{m-1})$$

$$S_{m-2}^{(q)} = 2^{-2}(\Phi(4q-4) + \Phi(4q-3) - \Phi(4q-2) - \Phi(4q-1)) \quad (q = 1,\ldots,2^{m-2})$$

. . . . . .

$$S_{m-l}^{(q)} = 2^{-l}\Big(\sum_{k=2^{l-1}+1}^{2^l} \Phi(2^l q - k) - \sum_{k=1}^{2^l-1} \Phi(2^l q - k)\Big) \tag{5.12}$$

$$(l = 1,2,\ldots,m-1 \quad ; \quad q = 1,2,\ldots,2^{m-l})$$

From Eq. (5.12) it can be seen that coefficient $S_{m-1}^{(q)}$ depends only on two consecutive values $\Phi(2q-2)$, $\Phi(2q-1)$ of $\Phi(z)$, and coefficient $S_{m-2}^{(q)}$ on four consecutive values $\Phi(4q-4)$, $\Phi(4q-3)$, $\Phi(4q-2)$, $\Phi(4q-1)$. In general, each of the coefficients $S_{m-l}^{(q)}$ depends on the behavior of $\Phi(z)$ on the interval $[2^l q-2^l, 2^l q-1)$. Therefore the expansion coefficients of $\Phi(z)$ in Haar series depend only on the local behavior of the function. As an example, the Haar expansion of the system described by Table 5.1 using equations (5.10) and (5.11), is as follows:

$$\Phi(z) = 1.5 - H_1^{(1)}(z) + H_1^{(2)}(z) - 0.5H_2^{(1)}(z) + 0.5H_2^{(2)}(z) - 0.5H_2^{(3)}(z) + 0.5H_2^{(4)}(z)$$

From Eq. (5.11) another expression can be derived to find spectrum of Haar functions which is actually the same equation as Eq. (3.12) but in a simpler formula (Eq. 5.13):

$$S = 2^{-m+l} H_m \Phi \tag{5.13}$$

where H_m is the Haar transform matrix whose elements can be calculated by Eq. (5.9), and Φ is the step function that defines our logical system. The important point about Eq. (5.13) is that the value of 2^{-m+l} is different for different rows of the matrix because l is different.

For example for the system described in Table 5.1 the spectrum of the Haar function can be calculated as follows:

Since m=3, the Haar transform matrix H_m using Eq. (5.9) or from Fig. 5.3 will be:

$$H_3 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}$$

and the spectrum of the function will be:

$$S = \begin{bmatrix} 2^{-3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2^{-3+0} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2^{-3+1} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2^{-3+1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2^{-3+2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2^{-3+2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2^{-3+2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2^{-3+2} \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 3 \\ 2 \\ 2 \\ 3 \\ 1 \\ 0 \end{bmatrix}$$

$$S = \begin{bmatrix} 2^{-3}(12) \\ 2^{-3}(0) \\ 2^{-2}(-4) \\ 2^{-2}(4) \\ 2^{-1}(-1) \\ 2^{-1}(1) \\ 2^{-1}(-1) \\ 2^{-1}(1) \end{bmatrix} \Rightarrow S = \begin{bmatrix} 1.5 \\ 0 \\ -1 \\ 1 \\ -0.5 \\ 0.5 \\ -0.5 \\ 0.5 \end{bmatrix}$$

From the spectrum vector S the Haar expansion will be:

$$\Phi(z) = 1.5 - H_1^{(1)}(z) + H_1^{(2)}(z) - 0.5H_2^{(1)}(z) + 0.5H_2^{(2)}(z) - 0.5H_2^{(3)}(z) + 0.5H_2^{(4)}(z)$$

which matches the result obtained before.

Haar expansion coefficients can also be calculated through a recursive method which has the minimum complexity (requiring minimum number of additions or subtractions) compared to other algorithms. If $\Phi(z)$ is the step function representing our Boolean system, the Haar spectrum of the function can be calculated as in Eq. (5.14):

$$S_{m-k}^{(q)} = 2^{-k} a_k(2^{m-k} - 1 + q) \tag{5.14}$$

where:

$$a_0(t) = \Phi(t) \qquad (t = 0, 1, \ldots, 2^m - 1)$$

$$a_k(t) = a_{k-1}(2t) + a_{k-1}(2t + 1) \qquad (t = 0, 1, \ldots, 2^{m-k} - 1)$$

$$a_k(2^{m-k} + t) = a_{k-1}(2t) - a_{k-1}(2t + 1) \qquad (k = 1, 2, \ldots, m)$$

The previous example can be solved by this method which gives the same result.

If in our original function the order of the arguments is changed, the Haar coefficients will change. For example, one can change the order of the arguments in such a way that the number of nonvanishing coefficients decreases. Karpovsky[6] has shown that because of the local behavior of the Haar functions, it is possible to find such an order of arguments that gives the minimum number of nonvanishing coefficients. The next section will discuss this issue.

5.4. Optimal Ordering of Arguments for Haar Expansions

The idea of image compression based on Haar transform comes from the fact that it is possible to find the optimal ordering of arguments which gives the minimum number of nonvanishing Haar coefficients. This property is due to the local behavior of Haar functions. In this section the algorithm to find such ordering of the arguments will be explained.

The objective is to find a matrix σ_{opt} in such a way that the number of Haar coefficients is minimum when the function is expanded over $z_{\sigma_{opt}} = \sigma_{opt} \otimes z$.
In the following the algorithm is clearly explained step by step:

1) For a system of k Boolean functions of m arguments $f^{(m-1)}(z)$, a characteristic function $f_i^{(m-1)}(z)$ $(i = 0, 1, \ldots, 2^k - 1)$ is defined as follows:

$$f_i^{(m-1)}(z) = \begin{cases} 1 & \text{if} \quad f^{(m-1)}(z) = i \\ 0 & \text{if} \quad f^{(m-1)}(z) \neq i \end{cases} \tag{5.15}$$

Table 5.2 shows the characteristic functions for our example.

2) An auto–correlation function $B_i^{(m-1)}(\tau)$ for each characteristic function $f_i^{(m-1)}(z)$ is defined as follows:

$$B_i^{(m-1)}(\tau) = \sum_{z=0}^{2^m-1} f_i^{(m-1)}(z)\, f_i^{(m-1)}(z \oplus \tau) \tag{5.16}$$

where $z \oplus \tau$ shows the modulo–2 bit by bit addition of z and τ bit by bit. Table 5.2 shows the auto–correlation functions for our example.

3) The values of $B^{(m-1)}(\tau)$ are calculated by Eq. (5.17) ,

$$B^{(m-1)}(\tau) = \sum_{i=0}^{2^k-1} B_i^{(m-1)}(\tau) \tag{5.17}$$

and the value of τ_{m-1} is obtained in such a way that it satisfies Eq. (5.18).

$$Max\ B^{(m-1)}(\tau) = B^{(m-1)}(\tau_{m-1}) \qquad where \qquad \tau \neq 0 \tag{5.18}$$

Table 5.2 shows $B^{(m-1)}(\tau)$ for our example, and it can be seen that τ_{m-1}=7.

4) A nonsingular matrix σ_{m-1} is determined so that:

$$\sigma_{m-1} \otimes \tau_{m-1} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \tag{5.19}$$

where $\otimes$ denotes modulo–2 multiplication.

Since in our example $\tau_{m-1} = \tau_2 = 7$, σ_{m-1} is determined as follows:

$$\sigma_{m-1} \otimes \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

z	τ	$f^{(2)}(z)$	$f_i^{(2)}(z)$				$B_i^{(2)}(z)$				$B^{(2)}(\tau)$	$f_{\sigma 2}^{(2)}(z)$
			$i=0$	1	2	3·	$i=0$	1	2	3·		
0	0	0	1	0	0	0	2	2	2	2	8	0
1	1	1	0	1	0	0	0	0	0	0	0	0
2	2	3	0	0	0	1	0	0	0	0	0	1
3	3	2	0	0	1	0	0	0	0	0	0	1
4	4	2	0	0	1	0	0	0	0	0	0	2
5	5	3	0	0	0	1	0	0	0	0	0	2
6	6	1	0	1	0	0	0	0	0	0	0	3
7	7	0	1	0	0	0	2	2	2	2	8	3

Table 5.2 The results of the first steps of the process

For this example σ_{m-1} can be:

$$\sigma_{m-1} = \sigma_2 = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

5) Construct the function $f_{\sigma_{m-1}}^{(m-1)}(z)$ in such a way that:

$$f_{\sigma_{m-1}}^{(m-1)}(z_{\sigma_{m-1}}) = f^{(m-1)}(z) \qquad \textit{and} \qquad z_{\sigma_{m-1}} = \sigma_{m-1} \otimes z \tag{5.20}$$

for our example $f_{\sigma_2}^{(2)}(z)$ is shown in Table 5.2.

6) The function $f^{(m-2)}(z)$, defined at 2^{m-1} points, is obtained as follows:

$$f^{(m-2)}(z) = f^{(m-1)}_{\sigma_{m-1}}(2z) + f^{(m-1)}_{\sigma_{m-1}}(2z + 1) \tag{5.21}$$

for our example $f^{(1)}(z)$ will be as shown in Table(5.3)

7) The above procedure is applied to $f^{(m-2)}(z)$, and the resulting matrix is called $\sigma^{(m-2)}$ which is of size $(m-1)\times(m-1)$. Now the matrix σ_{m-2} is defined as follows:

$$\sigma_{m-2} = \left[\begin{array}{c:c} \sigma^{(m-2)} & 0 \\ \hdashline 0 & 1 \end{array}\right] \tag{5.21}$$

For our example, the results of this step will be as in Table (5.3), where σ_1 is:

$$\sigma_1 = \left[\begin{array}{c:c} \sigma^{(1)} & 0 \\ \hdashline 0 & 1 \end{array}\right] = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

8) The above steps are repeated to find σ_{m-1}, σ_{m-2}, . . ., σ_1 . Then σ_{opt} is calculated by Eq. (5.22):

$$\sigma_{opt} = \sigma_1 \otimes \ldots \otimes \sigma_i \otimes \ldots \otimes \sigma_{m-2} \otimes \sigma_{m-1} \tag{5.22}$$

z	τ	$f^{(1)}(z)$	$B^{(1)}(\tau)$	$f^{(1)}_{\sigma 1}(z)$
0	0	0	4	0
1	1	2	0	4
2	2	4	0	6
3	3	6	0	2

Table 5.3 The results of the next steps of the process

where

$$\sigma_i = \left[\begin{array}{c:c} \sigma^{(i)} & 0 \\ \hdashline 0 & E_{m-i-1} \end{array}\right] \tag{5.23}$$

and E_{m-n-1} is the identity matrix of order *m–n–1*. So in our example we have:

$$\sigma_{opt} = \sigma_1 \otimes \sigma_2 = \left[\begin{array}{c:c} \sigma^{(1)} & 0 \\ \hdashline 0 & 1 \end{array}\right] \otimes \sigma_2 = \left[\begin{array}{ccc} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right] \otimes \left[\begin{array}{ccc} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{array}\right] = \left[\begin{array}{ccc} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{array}\right]$$

This was the last step of the algorithm to find σ_{opt} .

Now we can calculate $z_{\sigma_{opt}} = \sigma_{opt} \otimes z$ and expand our function over the new order of the arguments. Therefore for our example we have:

$$z_{\sigma_{opt}} = \sigma_{opt} \otimes z = \left[\begin{array}{ccc} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{array}\right] \otimes \left[\begin{array}{c} z^{(0)} \\ z^{(1)} \\ z^{(2)} \end{array}\right] = \left[\begin{array}{c} z^{(0)} \oplus z^{(2)} \\ z^{(1)} \oplus z^{(2)} \\ z^{(2)} \end{array}\right]$$

and therefore:

$$f(z_{\sigma_{opt}}) = \begin{bmatrix} 0 \\ 0 \\ 3 \\ 3 \\ 2 \\ 2 \\ 1 \\ 1 \end{bmatrix}$$

The spectrum of the function can then be calculated as follows:

$$S_{\sigma_{opt}} =$$

$$\begin{bmatrix} 2^{-3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2^{-3+0} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2^{-3+1} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2^{-3+1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2^{-3+2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2^{-3+2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2^{-3+2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2^{-3+2} \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 3 \\ 3 \\ 2 \\ 2 \\ 1 \\ 1 \end{bmatrix}$$

$$S_{\sigma_{opt}} = \begin{bmatrix} 1.5 \\ 0 \\ -1.5 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

It can be observed that the number of non–vanishing coefficients for the new ordering has been significantly decreased. The corresponding Haar expansion of the function will be as follows:

$$\Phi(z_{\sigma_{opt}}) = 1.5 - 1.5H_1^{(1)}(z_{\sigma_{opt}})$$

5.5. Image Compression

To compress image by the Haar transform, the pixel matrix is considered as our discrete function *f(z)*. Therefore the matrix can be converted to a vector (one dimensional matrix), and the same process as explained in the previous section can be performed. Therefore, the first step will be the calculation of σ_{opt} . Usually in black and white images each pixel value can be represented by 8 bits. In this case we only have 8 inputs, so σ_{opt} will be an 8×8 matrix. According to that the pixel elements are reordered, then Haar transform is taken, and the resulting vector usually has a much smaller number of nonzero elements. Then the zero elements can be coded by Run–Length coding, which in general needs a smaller number of bits to represent the data.

The work which has been done in this area [2] does not show how good this method is in the sense of the compression factor. They just show that the number of nonvanishing coefficients decreases significantly. This does not necessarily mean that the compression factor is significantly improved. The point is, we do not lose any information in this method of compression, in other words the reconstructed image is exactly the same as the original one.

Chapter 6

Image compression using Reed–Muller Transform

6.1. Introduction

In this chapter image compression based on Reed–Muller transform is investigated. First the basic topics needed as background are covered , then the compression method based on fixed polarity Reed–Muller transform is discussed.

6.2. The Galois field (2) algebra

Galois field algebra is actually a modulo–2 algebra, which has the following properties:

If $a,b,c \in \{0,1\}$, and $\oplus$ and $\odot$ represent modulo–2 addition and multiplication:

1) $a \oplus b \in \{0,1\}$ and $a \odot b \in \{0,1\}$

2) $a \oplus (b \oplus c) = (a \oplus b) \oplus c = a \oplus b \oplus c$ and

$a \odot (b \odot c) = (a \odot b) \odot c = a \odot b \odot c$

3) $a \odot (b \oplus c) = a \odot b \oplus a \odot c$

4) $a \oplus b = b \oplus a$ and $a \odot b = b \odot a$

5) $a \oplus 0 = a$ and $a \odot 1 = a$

6) $a \oplus a = 0$ and $a \odot a = a$

The first five properties are the same as boolean algebra's, but the sixth property reveals the differences between the two.

It can be seen for both algebra the operation of multiplication is the same, i.e. $a \odot b = a.b$, but the operation of addition is different so that we have $a \oplus b = \bar{a}.b + a.\bar{b}$. Substituting 1 for b in the latter equation gives $a \oplus 1 = \bar{a}$. Using de Morgan theory and the last two equations one can easily derive that $a+b = a \odot b \oplus a \oplus b$.

So we can observe all basic operations in Boolean algebra, i.e. addition, multiplication and negation, have equivalent in modulo–2 algebra. Therefore every Boolean function can be implemented over GF(2) too.

6.3. Reed–Muller Transform

Any switching function of n variables can be defined by 2^n coefficients in a sum of product form as Eq. (6.1):

$$f(x_0, x_1, ..., x_{n-1}) = d_0 \bar{x}_{n-1} \bar{x}_{n-2} ... \bar{x}_0 + d_1 \bar{x}_{n-1} ... \bar{x}_1 x_0 + ... + d_{2^n-1} x_{n-1} x_{n-2} ... x_0 \quad (6.1)$$

where $(d_0, d_1, d_2, ..., d_{2^n-1})$ are the coefficients of the products which represent the values in the output column of the truth table of the function. These coefficients can be represented in vector form D, called truth vector.

The function can also be represented by the Reed–Muller canonical form over Galois field (2) as Eq. (6.2):

$$f(x_0, x_1, ..., x_{n-1}) = a_0 \oplus a_1 x_0 \oplus a_2 x_1 \oplus a_3 x_0 x_1 \oplus ... \oplus a_{2^n-1} x_0 x_1 ... x_{n-1} \quad (6.2)$$

where $\oplus$ denotes modulo–2 addition and $(a_0, a_1, a_2, ..., a_{2^n-1})$ are the coefficients of the expansion. These coefficients can be represented in vector form A, called the function vector.

These two representations are related to each other by a transform matrix T as Eq. (6.3):

$$A = TD \tag{6.3}$$

For a function with three variables Eq. (6.3) is as follows:

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \end{bmatrix} \qquad \text{over GF(2)}$$

An important property of matrix T is that T is the inverse of itself, i.e. $T = T^{-1}$. So Eq. (6.4) is also true:

$$D = TA \tag{6.4}$$

Matrix T can be written in a recursive form as Eq. (5.5):

$$T = T_n = \begin{bmatrix} T_{n-1} & \mathbf{0} \\ T_{n-1} & T_{n-1} \end{bmatrix} \qquad \text{for n >= 1}$$

$$T_0 = 1 \tag{6.5}$$

So T_n can also be written as Eq. (6.6):

$$T_n = T_1 * T_{n-1} = T_1 * T_1 * T_1 * \cdots * T_1 \quad \text{n times} \tag{6.6}$$

Where * represents the Kronecker matrix product.

As an example consider the Boolean function $f(x_0,x_1,x_2) = \bar{x}_1 x_0 + x_2 x_1$, Which in terms of minterms will be $f(x_0,x_1,x_2) = \bar{x}_2\bar{x}_1 x_0 + x_2\bar{x}_1 x_0 + x_2 x_1 \bar{x}_0 + x_2 x_1 x_0$. From this function the truth vector is $A = [0\ \ 1\ \ 0\ \ 0\ \ 0\ \ 1\ \ 1\ \ 1]$. Now to find the Reed–Muller Canonical form of the function, T_3 must be calculated using Eq. (5.5). Then from Eq. (6.3), D can be calculated:

$$D = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad \text{over GF(2)}$$

$$= \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

So the RM canonical form of the function is $f(x_0,x_1,x_2) = x_0 \oplus x_0 x_1 \oplus x_1 x_2$.

6.4. Generalized Reed–Muller Transform

In the previous section the Reed–Muller expansion was explained and its form was shown as Eq. (6.2). In that form all the variables are in the positive form, while any variable x_i can be substituted with its negation $\bar{x}_i$, and still retain the canonical form. In the case that the variables are allowed to take negative polarities, we have the *Generalized Reed–Muller* (GRM) canonical form. And if each variable is restricted to retain the same polarity in all terms, i.e. either positive or negative but not both, the canonical form is called the *Fixed Polarity Reed–Muller* form.

For a function with n variables the number of possible arrangements of polarities is 2^n, so there are 2^n possible fixed polarity Reed–Muller forms. For a specific function the number of terms varies based on the polarity of the variables. For instance, in the previous example the Reed–Muller form of the function was found to be $f(x_0,x_1,x_2) = x_0 \oplus x_0x_1 \oplus x_1x_2$. If the polarity of the variables was selected as $\bar{x}_0x_1\bar{x}_2$, the result would be $f(x_0,x_1,x_2) = 1 \oplus \bar{x}_0 \oplus \bar{x}_0x_1 \oplus x_1\bar{x}_2$.

By finding the appropriate polarities one can reduce the number of terms of a function in fixed polarity RM form. The best polarity gives the least number of terms in a specific function.

6.5. Image Compression Using Fixed Polarity RM Transform

In the last section it was explained that if the best polarities for the variables are selected, the number of terms of each function will be reduced. Using this fact we are going to investigate if it can be used to compress the data of an image.

It was mentioned in the previous chapter that for a black and white picture, one byte is allocated for each pixel. So in a picture matrix each element consists of 8 bits which can be either 1 or 0. Therefore the picture matrix can be converted to 8 matrices from the most significant bit plane, to the least significant bit plane (each element of these matrices is either 1 or 0). The algorithm for the compression technique will be explained as follows:

```
for( i = Most significant bit plane to least significant bit plane )
    {
      for( each plane )
          {
              fetch a block of size ( N×N)
              find the best polarity of the variables
              Compute RM transform of the block
              use Run–Length coding to code the data of the transformed block
          }
    }
```

In this method each block of size (N×N) is considered as a Karnough map, so the number of 1s in each block shows the number of minterms. After finding its fixed polarity RM form with the minimum number of terms, which is done exhaustively [9], [10], [14], the new map usually contains fewer number of 1s. But there is an important point which is worth noting:

When Run–length coding is used, the compression ratio mostly depends on the number of transition elements in a bit plane matrix and not on the number of nonzero elements. For example in a 16 bit row of a block, it is possible to have 8 zeros and 8 ones with just one transition element as in Fig. 6.1 (a), alternatively it is possible to have 13 zeros and 3 ones with six transition elements as in Fig. 6.1 (b). Obviously the latter one will be less compressed than the previous one, while the number of nonzero elements in it is much less. So the pattern of the elements is also important, and just reducing the number of nonzero elements might not be sufficient to obtain a higher compression ratio.
As it will be explained in chapter 9, the compression factor using this method is not good, and the main reason is, what was mentioned above.

0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1

(a)

0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0

(b)

Fig. 6.1 The number of 1s in b is less, but the number of transition elements is more.

6.6. Comments on LSB planes

As mentioned before the least significant bit planes do not have as much effect on the values of pixels as the MSB planes. Therefore we can allow loss of information in these planes.

The simplest way to compress the data based on losing information is to consider some specific area and calculate if the number of 1s is more or less than that of zeros. If it is more, we change all the elements to 1, otherwise all the elements are changed to zero. For example in the LSB plane, areas of 5 bits can be specified. Then the above rule is applied.

Another method could be based on performing some small changes in the elements of each block of size NxN so that the resulting Karnough map would give a much smaller number of terms. Since finding the best fixed polarity RM form requires checking many different cases, and we do not know the desired polarity in advance, changing the elements will not work very good, unless changing the elements is based on a specific Reed–Muller form with predefined polarities. The latter method has not been attempted in this work.

6.7. Permutation of the elements of the Truth vector

In the following, the possibility of performing a permutation on the Reed–Muller transform is investigated.

It was explained that in a switching function the truth vector $\boldsymbol{D}$ is related to the function vector $\boldsymbol{A}$ by a transform matrix $\boldsymbol{T}$ as in Eq. (6.3). For example for a switching function with three variables the equation is as follows:

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \end{bmatrix} \qquad \text{over GF(2)}$$

From the above equation the local behavior of the Reed–Muller transform can be observed. It is obvious that only a_7 is globally sensitive, i.e. depends on all elements of the truth vector. The other elements of the function vector, i.e. a_0 to a_6, all are locally sensitive, which depend on a subset of the elements of the truth vector. Therefore the value of the elements of the function vector can be changed by a permutation of the elements of the truth vector. Therefore the best permutation of the elements of the truth vector can minimize the number of nonzero elements of the function vector.

As a result, a permutation matrix σ can be determined so that the permuted truth vector is given as Eq. (6.7):

$$D_\sigma = \sigma.D \tag{6.7}$$

Now the Reed–Muller transform of the permuted truth vector can be determined using Eq. (6.8):

$$A_\sigma = TD_\sigma \tag{6.8}$$

σ should be determined in such a way that $A\sigma$ contains the least number of 1s. D can be

reconstructed using Eq. (6.9):

$$\boldsymbol{D} = \sigma^{-1}\boldsymbol{T}^{-1}\boldsymbol{A}_{\sigma} \tag{6.9}$$

As an example assume that the truth vector of a switching function is $\boldsymbol{D}$=[0 0 1 1 0 1 0 1]. If the permutation matrix σ is chosen in such a way that the permuted truth vector is as $\boldsymbol{D}_{\sigma}$=[1 0 1 0 1 0 1 0], σ and the Reed–Muller transform of $\boldsymbol{D}_{\sigma}$ will be as follows:

$$\sigma = \begin{bmatrix} 0&0&0&1&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1&0 \end{bmatrix}$$

$$\boldsymbol{A}_{\sigma} = \begin{bmatrix} 1&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 1&0&1&0&0&0&0&0 \\ 1&1&1&1&0&0&0&0 \\ 1&0&0&0&1&0&0&0 \\ 1&1&0&0&1&1&0&0 \\ 1&0&1&0&1&0&1&0 \\ 1&1&1&1&1&1&1&1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad \text{over GF(2)}$$

It can be seen that the number of nonzero elements in $\boldsymbol{A}_{\sigma}$ is smaller than that in $\boldsymbol{D}$, and in general this method gives a much smaller number of nonzero elements compared to the method of the fixed polarity RM transform.

The problem with this method is how to determine the permutation matrix σ, and so far no method has been found. To find the best permutation it is required to check $\binom{n}{p}$

different cases, where n is the number of elements of the truth vector, and p is the number of nonzero elements in the truth vector. Therefore this method is not practical although it might give a very good result.

Chapter 7

A critique of the Reddy & Pai approach to Reed–Muller Image Compression

In this chapter the published paper on Reed–Muller image compression [3] is investigated. There are some mistakes in this paper which cause the results of this paper to be unacceptable.

7.1. The general method

In the paper by Reddy and Pai [3] the algorithm for image compression is as follows:

for (i = most significant bit plane to least significant bit plane) do
begin
for (each plane) do
begin
(a) fetch a block of size (n × n)
(b) compute the Reed–Muller transform of this block
(c) employ runlength coding on the resultant transform domain bit plane
end;
end;

As it can be seen, the body of the inner loop consists of three steps. There is nothing wrong with the first step (a), but the problems with step (b) and (c) cannot be ignored.

7.2. The problem with step (b)

In the paper the authors, Reddy & Pai, have not clearly explained what they have done, but from what they have written, it is obvious that it must be one of the following cases and nothing else.

1) The authors have just simply computed the positive polarity Reed–Muller transform of the block.

2) The authors have performed a permutation on the input sequence (as was explained in section 6.7), and then computed the positive polarity Reed–Muller transform.

In the first case we cannot get a good compression factor by just computing the Reed–Muller transform of the block. As a matter of fact, in this case even for the MSB plane the compression factor on the average is less than 1.6, which is not good at all. We can get a better compression factor even if we don't use a transformation, and just directly encode the plane.

In addition to that, in this thesis fixed polarity RM transform has been used, and although it is much better than the RM transform, still the result was not that good, so that the compression factor for the MSB plane is less than 2.

In the second case although the authors have mentioned something about permutation, they have not found a method to find the best permutation and still this problem is unsolved. The authors have clearly suggested this problem for those who are interested in this issue and want to work on that in the future. Therefore if they have applied a per-

mutation, it was done exhaustively which is not acceptable. It is obvious that finding the best permuted input sequence exhaustively cannot be realized by any hardware because it requires many cases to be checked, which is impossible.

7.3. Problems with step (c)

In this step there are some obvious mistakes, and in the following these mistakes are discussed:

The authors have used Relative Address Coding to code the data after performing the Reed–Muller transform.

The code used by Reddy & Pai is in Table 7.1. In the following, four problems with the data in this table have been explained in the order of their importance.

1) In chapter 3 Relative Address Coding has been explained in detail, and it was mentioned that in this method the relative address is computed either with respect to the transition elements in the current line or those in the previous line. And then for all runlengths a method of coding must be used. For example as it was shown before in the fixed polarity RM transform, Huffman coding was used which is the best method for coding. In the paper by Yamazaki [4] about RAC, other codes have been used which are not as good as Huffman codes but they take less memory space. In any case, all of these coding methods must satisfy the following condition:

" Each code must not be equal to a first part of another code. "

Huffman coding satisfies this condition and also the method used by Yamazaki does this too. Otherwise the codes cannot be detected.

Relative distance	Positive distance	Negative distance
1	00	10
2	01	11
3	0000	1010
4	0001	1011
5	0100	1110
6	0101	1111
7	000000	101010

Table 7.1 Codes used by Reddy & Pai for relative distances

Obviously the codes in Table 7.1 do not satisfy this condition, so they cannot be used to code the data. To make the problem clear assume a part of a block after Reed–Muller coding is as in Fig 7.1. According to Table 8.1 the code for element A must be 01 because the relative distance is +2, and the same code will be chosen for element B because the relative distance for B is +2 as well. Since A and B are adjacent transition elements, the code for them will be 0101. From the table it can be seen that the code for relative distance +6 is also 0101. Now if the receiver get the code 0101 it cannot identify if it is the code for the relative distance +6 or it is the code for two relative distance +2. This problem can be seen in all of the codes in this table.

One might say that the codes in Table 7.1 are just supplementary codes for the distances, and the first part of the codes are different (like what Yamazaki has done). But in this case according to Table 7.1 the codewords will be so long that not only does it not

compress the data, but it also creates more bits than what the original image needs.

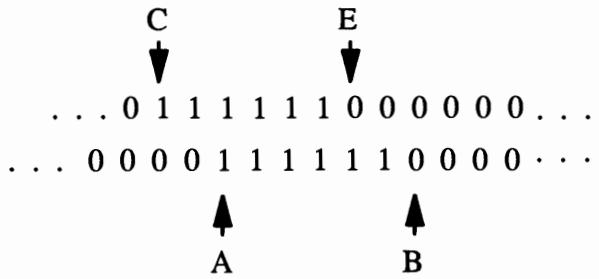


Fig. 7.1 An example showing contradiction in Reddy & Pai coding

2) The authors have mentioned that depending on the probability of the occurrence, the RAC distances (0, +1, –1) are given special code words. According to Table 7.1 the authors have defined the codewords for +1 and –1 but not for 0. Anyway, the obvious thing is that all the codes must be predefined. It would be impractical to first find the probability of the occurrence of the runlengths for every single matrix of an image and then according to that, find the codewords for them.

Another important problem is that usually the most common relative distance is 0, so it should have the shortest codeword, which in most of the cases consists of just one bit, i.e. either 0 or 1, or two bits. But according to Table 7.1 it is impossible to have a short codeword for relative distance 0, because in this case the condition explained previously cannot be satisfied.

3) Since the size of the blocks has been chosen to be 16×16, when the RAC method is used, there is a need to define the codes for at least 16 positive relative distances, and not just for 8. From Table 7.1 it seems that the authors have not understood the RAC method sufficiently well. Probably they have thought that since the distance of each transition

element from one of the row ends is less or equal to 8, it is enough to just define 8 positive relative distances. But it is impossible to decode the data this way, and 16 positive relative distance have to be defined, although for negative relative distance 8 codes are enough.

4) In Relative Address Coding, as mentioned in chapter 3, sometimes we need to compute the relative distance with respect to the transition elements in the current line, but according to Table 7.1 it seems that the authors of this paper have not defined them. In any case there are two possibilities:

a) They have defined them and used them but they just did not mention it in the paper. In this case, either their codes are all wrong like those in Table 7.1, or the authors use very long codewords, because the condition mentioned for the coding should not be contradicted at least for the rest of the codewords.

b) They have not defined the relative distances, and so have not used any codewords for the relative distance in the current line. In this case the coding will not be optimum. And the method is not the RAC method any more. Some modification has to be done to make the method work at the price of a decreased compression factor.

From the above discussion it is obvious that the reconstruction of the image is impossible with this type of coding.

The final point about this paper is that the quality of the picture is not that good even with the compression factor of 2.5 bit per pixel. It seems that there is a loss of information in more than 4 bit plane matrices to get 2.5 bit per pixel.

7.4. Conclusion

This paper has given a new idea for image compression, although the method proposed by the authors is not good. Unfortunately, many crucial mistakes in this paper cause the readers not to be able to evaluate the quality and usefulness of the idea of using Reed–Muller transform for image coding.

Chapter 8

Row Xoring And Plane Xoring Algorithm

8.1. Introduction

In this chapter a method is introduced which takes a good advantage of the correlation between the adjacent pixels. This method is based on Xoring the adjacent lines and the adjacent planes. The process is very fast with a much better compression factor compared to the method based on the Reed–Muller transform.
In the following, first the compression based on Xoring the lines, and then the compression based on Xoring the lines and the planes will be explained.

8.2. Compression using Xoring the adjacent lines

In most images the values of the adjacent pixels are highly correlated. Therefore most of the time the value of a pixel in a line is very close to its adjacent pixel at the adjacent line. Because of this fact, when a picture matrix is converted to the eight matrices, from the most significant bit plane to the least significant bit plane, the information of the two adjacent lines in MSB planes is very much the same, and as we go towards LSB planes the correlation becomes lower. Therefore it can be seen that the properties of the bit planes are different and we should treat them differently.

We can summarize the behavior of the bit planes from MSB planes towards LSB planes as follows:

1) In MSB planes the data of adjacent lines are very similar, so we can also take advantage of that to compress the data. But in the LSB planes the information of adjacent lines is not that much similar so that in the LSB and the second LSB planes, the information can be assumed random, and there is almost no correlation between them (specially in pictures from nature).

2) Any change in the value of elements in the MSB planes will change the value of the corresponding pixel greatly. Therefore for MSB planes the effort should be taken not to lose any information while compressing the data. But changes in the values of elements in LSB planes do not have much effect on values of the corresponding pixels, so that changes in the LSB and the second LSB planes are almost invisible to us.

From the above discussion we can develop a method to improve the data compression. We know that adjacent lines in MSB planes are very similar specially in the MSB and the second MSB planes. Therefore by Xoring the adjacent lines bit by bit, the resulting line will contain many 0s and few 1s, so that the number of nonzero elements and also the number of transition elements are substantially decreased. If this method is used for the first two MSB planes the result will be much better than by using the fixed polarity RM transform, because if the fixed polarity RM transform is used, although in most cases the number of nonzero elements is decreased, the number of transition elements is much higher comparing to the case of Xoring the lines.

Therefore, for the Most Significant Bit planes, the following algorithm can be used to improve the compression factor:

```
for( i = Most Significant bit planes )  {
    for( each plane )  {
        fetch a Row
        if( first Row)
            put it in the first Row of the new plane without changing
        else
            Xor the fetched Row with the preceding Row bit by bit and put in
the new plane
        }
    Use Run–Length coding to code the data of the resulting plane
}
```

There are some points on Run–Length coding that should be noted. As mentioned before, the properties of the bit planes for the same picture are different so they should be treated differently. If RLC is used to code the data, different Huffman code tables should be defined for different planes. For example in the MSB plane of a picture there exist long runs even with more than 200 bits while in the second MSB plane the number of such runs is much smaller, and in the third MSB plane one can hardly find runs longer than 100 bits. Huffman codes for the first four MSB planes are shown at the end of Chapter 9. These codes have been defined based on the information from 10 different pictures. The use of different Huffman code tables requires more memory space but in exchange improves the compression factor.

The method that has been used for RLC for our bit planes is a little different from the method explained in Chapter 3. In this method the Huffman code table used for zero runs is the same as the table used for One Runs. The only thing that distinguishes the Zero Runs from the One Runs is that we assume every row of a matrix to start with a Zero Run, and for the case that a row starts with a One Run it is assumed that it starts with a Zero Run of length zero. Assuming that the first run is a Zero Run, the type of the rest of the runs in the line is clearly determined.

Also, an End Of Line (EOL) code can be inserted at the end of each line for line synchronization. This reduces compression factor slightly but in exchange if some error happens in the transmission line, EOL causes this error not to be distributed to the whole data, and just this single specific line is affected.

8.3. An improvement by Xoring the planes

In the previous method it has been taken advantage of the fact that the adjacent pixels are similar. But we can take more advantage of this fact.

It can be experimentally shown that in most of the pictures the difference between the adjacent pixels in more than 95% of the cases is less than 16. Knowing this fact we can improve the previous method.

For instance, let us assume the previous method has been used, and the planes with Xored lines are ready. Now considering that the difference between the adjacent pixels is usually less than 16, we can see if an element in the 3rd MSB plane (with the exception of the first line) is one, the corresponding elements in the 4th MSB plane will be in most cases one as well. And if an element in the 2nd MSB plane is one the corresponding ele-

ments in the 3rd and the 4th MSB planes in most of the cases are one. And if an element in the MSB plane is one, the corresponding elements in the 2nd, 3rd and 4th MSB planes in most of the cases are one. Therefore it is a good idea to Xor the resulting planes after xoring the lines, because the number of 1s and the number of transition elements will decrease.

Tables 8.1 and 8.2 show the above issue more clearly. The five different cases that are listed in Table 8.1 happen in most of the cases, and after Xoring the planes the results will be as in Table 8.2. It can be clearly seen that the number of 1s and therefore the number of transition elements are both decreased.

	Case	MSB	2nd MSB	3rd MSB	4th MSB
Corresponding elements in the planes after xoring the lines	#1	0	0	0	0
	#2	0	0	0	1
	#3	0	0	1	1
	#4	0	1	1	1
	#5	1	1	1	1

Table 8.1 Five cases which occur most of the time

Therefore the following procedure can be used to obtain a better compression factor comparing to the previous methods:

	Case	MSB	2nd MSB	3rd MSB	4th MSB
Corresponding elements in the resulting planes after xoring the planes with Xored lines	#1	0	0	0	0
	#2	0	0	0	1
	#3	0	0	1	0
	#4	0	1	0	0
	#5	1	0	0	0

Table 8.2 The result from Table 8.1 after Xoring the planes

```
for( i = Most Significant bit planes )table 5.1
  {
    for( each plane )
      {

        fetch a Row
          if( first Row)
            put it in the first Row of the new plane without changing;
          else
              Xor the fetched Row with the preceding Row bit by bit and put in
           the new plane;
      }
  }
for( i = new planes with Xored lines )
  {
```

```
    if ( ! first plane)
        {
            Xor each element with corresponding element of the previous
        matrix and put the results in a new matrix;
            Use Run–Length coding to code the data of the resulting planes;
        }
}
```

The experimental results (next chapter) show that this method decreases the number of transition elements by 20–30%, and increases the compression factor by almost the same factor.

For LSB planes, the same method, explained in chapter 6, can be used.

Chapter 9

Discussion of the experimental Results

In this chapter the simulation results of the two methods (fixed polarity Reed–Muller transform and Xoring the lines and planes) are discussed. For the method based on Reed–Muller transform the size of the blocks has been taken to be 16x16.

In Table 9.1 compression factors of the four MSB planes from the two methods are shown. These factors have been calculated based on 10 different pictures of natural scenes. Also the number of transition elements for a typical image in both cases are shown. As we see the number of transition elements in the case of using Reed–Muller transform is much higher than in the case of using Xoring the lines, so the compression ratio for the former case is lower.

The data in this table are based on the fact that there is no any information loss after compression of the bit planes.

There are some points about Huffman coding that should be explained:

1) For the case of Xoring, Huffman codes have been computed for the MSB and the 2nd MSB planes separately from those of the 3rd MSB and 4th MSB planes (see the end of the chapter), and each Huffman table contains two types of codewords: Terminating Codewords (TC) and Make Up Codewords (MUC). For each bit plane, the runs between 0 and N are transmitted using a single terminating codeword, and the runs longer than N bit are transmitted by a MUC and a TC.

For the case of MSB and 2nd MSB, N is taken as 256. .

For the case of 3rd MSB and 4th MSB, N is taken 64.

Bit Plane		MSB	2nd MSB	3rd MSB	4th MSB
Compression Factor	Xoring with RLC coding	3.9	2	1.4	1.15
	Reed–Muller Transform with RAC	1.9	1.2	1	1
# of Transition Elements For a typical image 256×256	Xoring	4000	9300	15500	21500
	Reed–Muller Transform	11000	19000	25500	29000

Table 9.1 The comparison of Fixed Polarity Reed–Muller compression and the compression using Xoring the lines

The reason for the above assignment is that as we start from MSB to LSB, the number of long runs becomes smaller and smaller. For example in the 4th MSB planes most of the runs (more than 95%) are less than 64 bits so it would be waste of memory space if we define the Huffman codes for 256 bits, and the improvement would not be that much.

2) In the case of Reed–Muller transform, like in the previous case, Huffman codes have been computed for MSB and the 2nd MSB separately from the 3rd MSB and 4th MSB.

9.1. Evaluation of image compression based on fixed polarity Reed–Muller Trans form

As mentioned before, there are two major things that stop us from obtaining the maximum compression factor, while Run–Length coding. In the following, this issue will be explained in more detail.

1) As explained before, Huffman coding gives an integer number of bits for each codeword r, and it is equal to or greater than $-\log_2 P(r)$. This causes the compression factor to be decreased.

As an example assume that there are just three different Runs of 0, 1 and 00 in some bit planes. If the probability of the occurrence of the Run 0 is not less than the other two, the Huffman code will be something as follows:

Run	Huffman code
0	0
1	10
00	11

Obviously in this case it would be much better if Huffman coding were not used because the compression factor will be less than one. In other words not only does the data not get compressed, but there will be more bits after coding. In the case that the probability of the occurrence of each code is equal, i.e. 1/3, the compression factor would be 4/5=0.8, and the reason for such bad results is that two of the Runs have to be coded by two bit codes.

2) The Huffman codes are defined based on the probability of the occurrence of Run–Lengths over many typical documents and it is not necessarily optimum for each specific case.

For example assume in some bit planes there are just four different Runs of 000, 111, 001 and 100, and based on the statistics averaged on all the planes, the Huffman codes are 0, 10, 110 and 111 respectively. Now if in a specific plane the probability of the occurrence of the runs are equal, i.e. %25, the predefined Huffman codes are not the best coding, and the increase in the number of bits required to compress the data comparing to the best Huffman codes will be equal to:

$$\frac{(0.25 \times 1) + (0.25 \times 2) + (0.25 \times 3) + (0.25 \times 3)}{4 \times (0.25 \times 2)} = 1.125$$

which shows 12.5% increase.

The problem with fixed polarity RM transform is that it cannot substantially decrease the number of transition elements, specially for the case of 3rd MSB toward LSBs the number of transition elements is still too high, and that is because of the fact that in fixed polarity RM forms just 2^n different cases are checked, where **n** is the number of variables. In other words since in fixed polarity RM neither the permutation of the variables nor the inconsistent polarity of the variables is allowed, a substantial decrease in the number of terms cannot be expected.

In general if $\mathbf{n_t}$ is the number of transition elements in a bit plane, and $\mathbf{H_t}$ is the average number of bits needed to code a run in the bit plane, the compression factor will be greater than one if the product of $\mathbf{n_t} \times \mathbf{H_t}$ is less than the number of elements of the plane. For

example, for a 256×256 image, each plane contains 65536 bits, so in order to have the compression factor greater than one for each plane, $\mathbf{n_t} \times \mathbf{H_t}$ must be less than 65536. In the case of using Reed–Muller transform usually the number of transition elements in each plane from the 3rd MSB to LSBs is higher than 20000. So $\mathbf{H_t}$ must be about 3 or less. But because of what was mentioned above, $\mathbf{H_t}$ is not very low, so that usually for these planes the compression factors are very low and sometimes less than 1. In other words the number of bits needed to code a run on the average is longer than the average length of the run, i.e. the numerator of the term in the right side of Eq.(9.1) becomes less than the denominator.

$$Q_{max} = \frac{\bar{r}_1 + \bar{r}_0}{H_1 + H_0} \tag{9.1}$$

9.2. Examples

In this section simulation results on three different pictures are shown. Table 9.2 shows the compression factor for the first four MSB planes. As it can be seen, we can get a significant improvement by Xoring the planes, and the quality of the images with compression factor 8/2.5 is very good. It should be noted that the following pictures have been printed by laser printer, so they are not the accurate representations of the images as shown on a gray–scale screen. Subjective quality estimates are the only effective technique discovered to date for comparing images.

	Image		MSB	2nd MSB	3rd MSB	4th MSB
Xoring the lines	Lady	# of transition elements	4021	9216	15455	21460
		# of bits after compression	17296	32627	47034	57547
		Compression factor	3.8	2.0	1.4	1.1
	Rose	# of transition elements	3496	8402	13405	20625
		# of bits after compression	15731	31582	43377	55668
		Compression factor	4.2	2.1	1.5	1.2
	House	# of transition elements	3396	7312	13294	20342
		# of bits after compression	14077	26080	40774	53634
		Compression factor	4.7	2.5	1.6	1.2
Xoring the lines + Xoring the planes	Lady	# of transition elements	4021	6499	10290	14018
		# of bits after compression	17296	24434	34646	42536
		Compression factor	3.8	2.7	1.9	1.5
	Rose	# of transition elements	3496	5746	7589	14019
		# of bits after compression	15731	23529	31507	44924
		Compression factor	4.2	2.8	2.1	1.5
	House	# of transition elements	3396	5404	10528	16053
		# of bits after compression	14077	20408	35578	46345
		Compression factor	4.7	3.2	1.8	1.4

Table 9.2 The effect of Xoring the planes after Xoring the lines

Fig. 9.1 Original image (8 bits per pixel)

Fig. 9.2 Compressed image (2.5 bits per pixel)

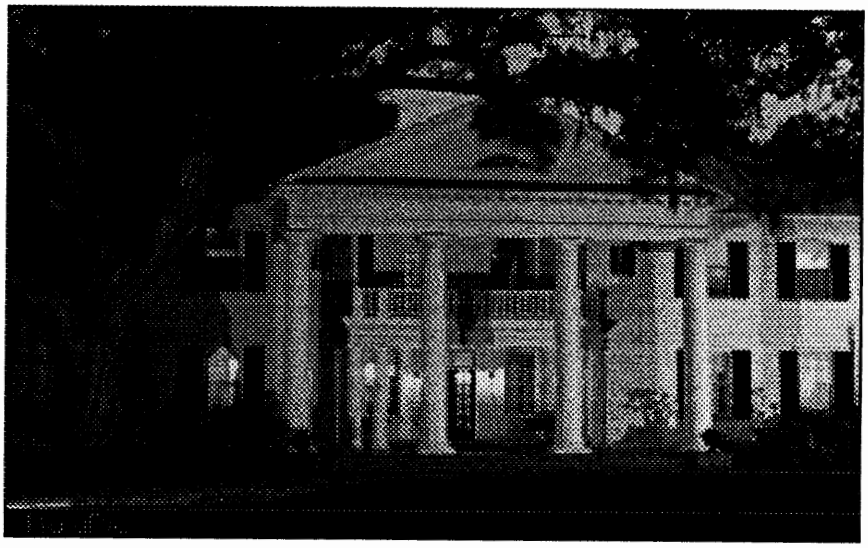


Fig. 9.3 Original image (8 bits per pixel)

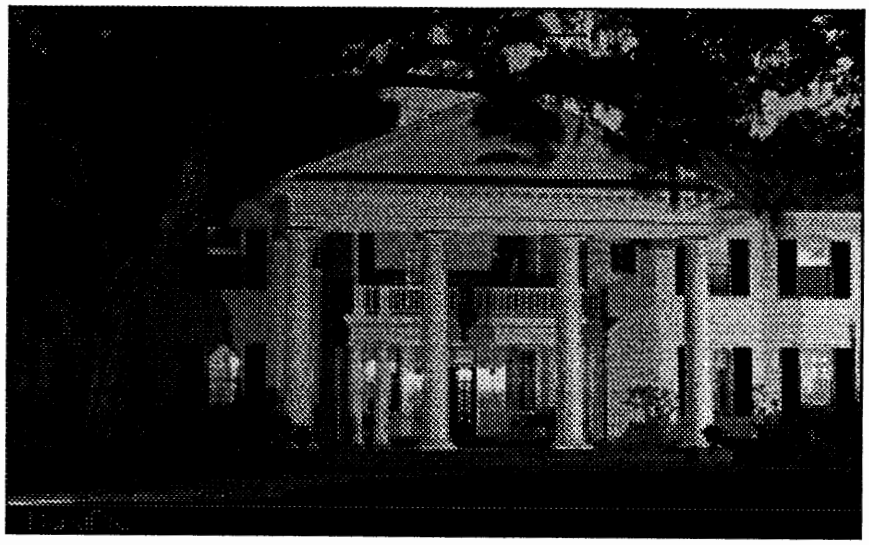


Fig. 9.4 Compressed image (2.5 bits per pixel)

Fig. 9.5 Original image (8 bits per pixel)

Fig. 9.6 Compressed image (2.5 bits per pixel)

9.3. Evaluation of image compression method based on Xoring the lines and the planes

The most important advantage of the method based on Xoring the lines and the planes is its simplicity which ultimately causes the process to be fast. The only logic gate needed for the mapping process is Exclusive–OR gate. And since $a \oplus b = c \Rightarrow b = a \oplus c$, to reconstruct the matrix, again just Exclusive–OR is needed..

Table 9.3 shows the comparison between the method based on JPEG algorithm and that based on the Xoring the lines and the planes.

	Speed	Compression Factor	Complexity
JPEG	Slow	Excellent	Complicated
Xoring	Fast	Good	Simple

Fig. 9.3 The comparison between the JPEG algorithm and Xoring The lines and the planes.

The mapping process requires just 2 Exclusive–ORs for each bit of the bit planes. Therefore for each pixel it requires 16 XOR gates, while for the case of the JPEG, the algorithm explained in **4.2.1** shows that for each pixel, an expression containing one addition and two multiplications needs to be executed 64 times.

Therefore the mapping process for the case of JPEG is much more complicated than that for the case of Xoring the lines and the planes, and this makes the JPEG Process much slower than Xoring the lines and the planes.

The most important advantage of the JPEG algorithm is the fact that with JPEG it is possible to obtain a very high compression factor, specially for the continuous–tone images. Xoring the lines and the planes to obtain a high compression factor (more than 5), requires loss of information from the 4th and sometimes even the 3rd MSB planes which has a bad effect on the quality of the picture. However, figures 9.2, 9.4, 9.6 shows that we can have a good quality images with compression factor 8/2.5. Figures 9.7, 9.8 and 9.9 show the compressed images with compression factor 8/2.5 using JPEG.

Fig. 9.7 Compressed image using JPEG
(2.5 bits per pixel)

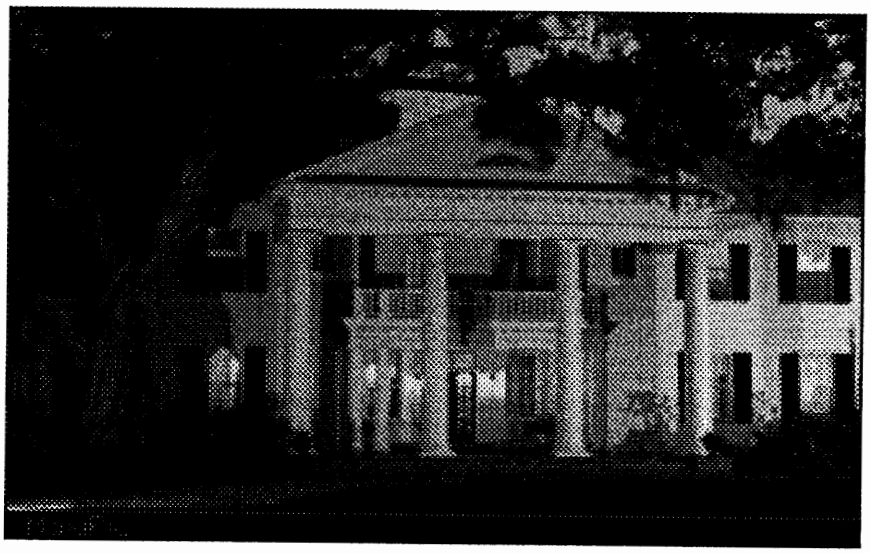


Fig. 9.8 Compressed image using JPEG (2.5 bits per pixel)

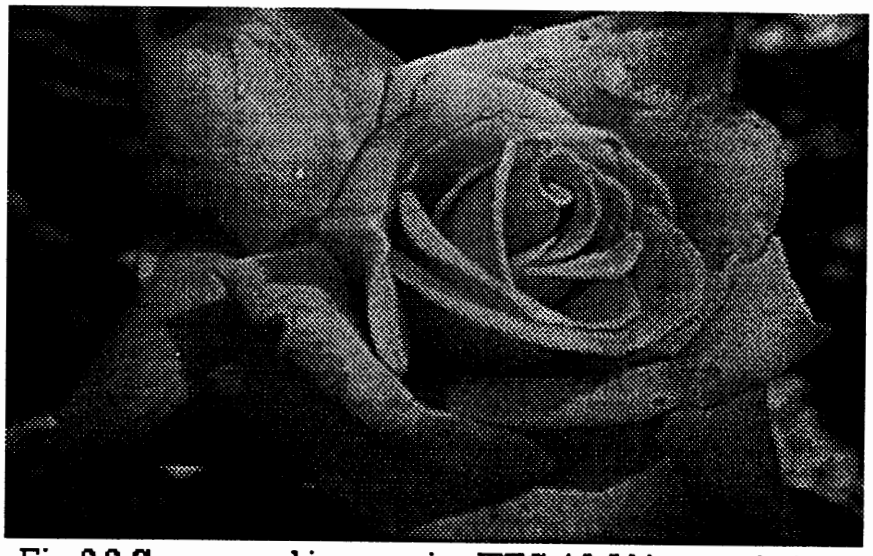


Fig. 9.9 Compressed image using JPEG (2.5 bits per pixel)

table 9.3: Huffman codes used for the runs in the MSB and the 2nd MSB planes after Xoring the lines.

# of bits	Huffman–code
0	000011100
1	1
2	011
3	0011
4	00011
5	01011
6	001001
7	010011
8	010101
9	0000101
10	0000011
11	0010110
12	0010111
13	0100011
14	0101001
15	00001111
16	00000101
17	00000001
18	00010011

# of bits	Huffman–code
19	00010101
20	00101001
21	00101011
22	01001001
23	01001011
24	01000101
25	000011101
26	000011011
27	000011001
28	000010011
29	000010010
30	000000111
31	000100011
32	000100001
33	000100101
34	000100010
35	000100100
36	000101001
37	000101111

# of bits	Huffman–code
38	000101101
39	000101110
40	001000101
41	001000001
42	001010001
43	001010100
44	001010101
45	010001001
46	010000111
47	010000101
48	010000110
49	010000011
50	010100001
51	010000001
52	010100011
53	0000110101
54	0000100011
55	0000100001
56	0000010001

# of bits	Huffman–code
57	0000000011
58	0000001100
59	0000001101
60	0001000001
61	0001010001
62	0001011000
63	0001011001
64	0010001111
65	0010001001
66	0010000100
67	0010000101
68	0010000110
69	0010000111
70	0010000001
71	0010100000
72	0010100001
73	0100101011
74	0100101010
75	0100001000

# of bits	Huffman–code
76	0100001001
77	0100000101
78	0100000001
79	0101000001
80	0101000100
81	0101000101
82	00001100011
83	00001101001
84	00001000001
85	00001000100
86	00001000101
87	00000100101
88	00000100110
89	00000100111
90	00000100001
91	00000100000
92	00000100100
93	00000010110
94	00000010111

# of bits	Huffman–code
95	00000010101
96	00010000001
97	00010100001
98	00100011011
99	00100011100
100	00100011101
101	00100010001
102	00100011001
103	00100011010
104	00100000001
105	01001010011
106	01001000001
107	01001010000
108	01001010001
109	01001010010
110	01000100010
111	01000100011
112	01000001001
113	01000100000

# of bits	Huffman–code
114	01000100001
115	01010000000
116	01010000001
117	01000000001
118	000011010001
119	000011000101
120	000010000001
121	000011000000
122	000011010000
123	000011000001
124	000011000010
125	000011000011
126	000011000100
127	000000100110
128	000000100111
129	000000101000
130	000000101001
131	000100000001
132	000000100010

# of bits	Huffman–code
133	000000100011
134	000000100100
135	000000100101
136	001000100001
137	001000110000
138	001000110001
139	000101000001
140	000101000000
141	010010001001
142	010010001010
143	010010001011
144	010010001100
145	010010001101
146	010010001110
147	010010001111
148	010010000001
149	010010000000
150	010010000100
151	010010000101

# of bits	Huffman–code
152	010010000110
153	010010000111
154	010010001000
155	010000010001
156	010000000001
157	010000000000
158	010000010000
159	0000001000001
160	0000001000010
161	0001000000001
162	0000001000011
163	0000000001011
164	0000000001001
165	0000000001010
166	0000001000000
167	0010000000010
168	0010001000000
169	0010001000001
170	0010000000011

# of bits	Huffman–code
171	0010000000000
172	0010000000001
173	00000000011100
174	00000000011101
175	00000000011110
176	00010000000001
177	00000000011111
178	00000000000001
179	00000000010000
180	00000000010001
181	00000000011000
182	00000000011001
183	00000000011010
184	00000000011011
185	000000001000000
186	000000001000001
187	000000001000010
188	000000001000011
189	000000001000100

# of bits	Huffman–code
190	000000001000101
191	000000001000110
192	000000001000111
193	000000001001000
194	000000001001001
195	000000001001010
196	000000001001011
197	000000001001100
198	000000001001101
199	000000001001110
200	000000001001111
201	000000001010000
202	000000001010001
203	000000001010010
204	000000001010011
205	000000001010100
206	000000001010101
207	000000001010110
208	000000001010111

# of bits	Huffman–code
209	000100000000000
210	000000001011000
211	000000001011001
212	000000001011010
213	000000001011011
214	000000001011100
215	000000001011101
216	000000001011110
217	000000001011111
218	000000000001111
219	000000000010000
220	000000000010001
221	000000000010010
222	000000000010011
223	000000000010100
224	000000000010101
225	000000000010110
226	000000000010111
227	000000000011000

# of bits	Huffman–code
228	000000000011001
229	000000000011010
230	000000000011011
231	000000000011100
232	000000000011101
233	000000000011110
234	000000000011111
235	000000000000001
236	000000000001110
237	000000000000100
238	000000000000101
239	000000000000110
240	000000000000111
241	000000000001000
242	000000000001001
243	000000000001010
244	000000000001011
245	000000000001100
246	000000000001101

# of bits	Huffman–code
247	000010000000000
248	000010000000001
249	000010000000010
250	000010000000011
251	000010000000100
252	000010000000101
253	000010000000110
254	000010000000111
255	0000000000000000
256	0000000000000001

table 9.4: Huffman codes used for the runs in the 3rd MSB and the 4th MSB planes after Xoring the lines.

# of bits	Huffman–code
0	010000100
1	1
2	011
3	001
4	0001
5	01011
6	010011
7	010001
8	010101
9	000001
10	0100101
11	0101001
12	0101000
13	0000101
14	0000111
15	01001001
16	01000011
17	01000001
18	00000001

# of bits	Huffman–code
19	00001101
20	00001100
21	010000101
22	000000111
23	000000101
24	000000001
25	000010011
26	000010010
27	000010001
28	000010000
29	0100100011
30	0100000011
31	0000001101
32	0000001100
33	0000000001
34	01001000001
35	01001000011
36	01001000101
37	01000000101

# of bits	Huffman–code
38	01000000001
39	01000000011
40	01000000010
41	00000010001
42	00000010011
43	00000010010
44	00000000001
45	010010000001
46	010010000101
47	010010001001
48	010010000100
49	010010001000
50	010000000001
51	000000100001
52	000000100000
53	000000000001
54	0100100000001
55	0100100000000
56	0100000010011

# of bits	Huffman–code
57	0100000000001
58	0100000010000
59	0100000010001
60	0100000010010
61	0100000000000
62	0000000000001
63	0000000000000000
64	0000000000000001
128	0000000000000010
192	0000000000000011
256	0000000000000100
320	0000000000000101
384	0000000000000110
448	0000000000000111

Chapter 10

Conclusion

An image compression method based on fixed polarity Reed–Muller Transform was created and analyzed in order to improve the method introduced by Reddy & Pai [3]. It was shown that this method cannot provide a good compression factor, therefore it cannot be a good candidate for image compression, although the method is fast.

It was also shown that, the paper published by Reddy & Pai on Reed–Muller image compression contains several errors which make the paper invalid.

A fast algorithm for image compression based on Xoring the adjacent elements of the bit planes and Xoring the corresponding bits of the resulting planes was introduced, and it was demonstrated that this method provides a much better compression factor than the method based on Reed–Muller transform, so that the quality of the reconstructed image for the compression factor of 8/2.5 is very good. Remarkably, in addition to high compression factor the hardware to realize this technique is very simple, and requires the minimum number of operations, comparing to the other methods.

If an efficient algorithm is developed for optimum permutation of the input sequence in Reed–Muller image compression, the result might be good, but nobody has developed any method yet. Therefore for future work it can be a subject to work on.

REFERENCES

1. Hunter, R., Robinson, A. H. International digital facsimile coding standards. In *Proceedings of the IEEE,* Vol. 68, No. 7, July 1980, pp. 854–867.

2. Reddy, B. R. K., Siddiqi, M. U., Mullick, S. K. Image data compression using local behavior of Haar Transform. In *J. INSTN. ELECTRONICS & TELECOM. ENGRS.,* Vol. 29, No. 5, 1983, pp. 204–210.

3. Reddy, B. R. K., Pai, A. L., Reed–Muller Transform Image Coding, *Computer Vision, Graphics, and Image Processing,* Vol. 42, 1988, pp. 48–61.

4. Yamazaki, Y., Wakahara, Y., Teramura, H. Digital Facsimile equipments "Quick–FAX" using a new redundancy reduction technique. In *National Telecommunications Conference.* 1976, pp. 6.2.1–6.2.5

5. Huffman, D. A. A method for the construction of minimum redundancy codes. In *Proceedings IRE,* Vol. 40, 1962, pp.1098–1101.

6. Karpovsky, M. G. Finite orthogonal series in the design of digital devices, *Halsted Press,* 1976.

7. Reed, I. S. A class of multiple–error correcting codes and their decoding scheme, In *IRE Transactions on Information Theory,* Vol. PGIT–4, 1954, pp. 38–49.

8. Muller, D. E., Application of Boolean algebra to switching circuit design and to error detection, In *IRE Transactions on Electronic Computers,* Vol. EC–3, September 1954, pp. 6–12.

9. Green, D. H., Modern Logic Design, Electronic Systems Engineering Series, 1986.

10. Zhang, Y. Z., Rayner, P.J.W., Minimization of Reed–Muller polynomials with fixed polarity, In *IEE proceedings,* Vol. 131, Pt. E, No. 5, September 1984, pp. 177–186.

11. Duan, J. R., Wintz, P. A., Information Preserving Coding for multi spectral scanner data, TR–EE–74–15, School of Electrical Engineering, Purdue University, Indiana, 1974.

12. Chen, P. H., Wintz, P. A., Data compression for satellite images, TR–EE–76–9, School of Electrical Engineering, Purdue University, Indiana, 1976.

13. Gonzales, R. C., Wintz, P. A., Digital Image Processing, *Addison–Wesley Publishing Company,* 1983.

14. Sarabi, A., Perkowski, M. A., Cube based method for optimal and quasi–optimal minimization of Consistent Generalized Reed–Muller expansions, School of electrical Engineering, Portland State University, 1992.

15. Wallace, G. K., The JPEG still picture compression standard, Multimedia Engineering Digital Equipment Corporation, 1991.

16. Nelson, M., The Data Compression Book, Redwood City, CA: M&T Books, 1991.